HZ BOOKS



Professional

# Assembly Language

# 汇编语言程序设计

（美） Richard Blum 著

马朝晖 等译

机械工业出版社
China Machine Press



Professional

# Assembly Language

# 汇编语言程序设计

(美) Richard Blum 著

马朝晖 等译

机械工业出版社
China Machine Press

每种高级语言程序在连接为可执行程序之前，都必须被编译为汇编语言程序，因此对于高级语言程序设计者来说，了解编译器如何生成汇编语言代码十分有用。

本书分为三部分。第一部分讲解汇编语言程序设计环境基础，第二部分研究汇编语言程序设计，最后一部分讲解高级汇编语言技术。本书的主要目的是向使用高级语言的程序员讲解编译器如何从C和C++程序创建汇编语言例程，以及编程人员应如何掌握生成的汇编语言代码，调整汇编语言例程以提高应用程序的性能。

本书适合有一定编程经验的开发人员参考。

Richard Blum: Professional Assembly Language (ISBN: 0-7645-7901-0).

Authorized translation from the English language edition published by John Wiley & Sons, Inc.

Copyright © 2005 by Wiley Publishing , Inc.

All rights reserved.

本书中文简体字版由约翰-威利父子公司授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

**版权所有，侵权必究。**

**本书法律顾问　北京市展达律师事务所**

**本书版权登记号：图字：01-2005-1622**

**图书在版编目（CIP）数据**

汇编语言程序设计/（美）布鲁姆（Blum, R.）著；马朝晖等译. -北京：机械工业出版社，2006.1

书名原文：Professional Assembly Language

ISBN 7-111-17532-8

Ⅰ.汇 … Ⅱ.① 布 … ② 马 … Ⅲ.汇编语言-程序设计 Ⅳ.TP313

中国版本图书馆CIP数据核字（2005）第117128号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：赵　康　刘立卿

北京瑞德印刷有限公司印刷 · 新华书店北京发行所发行

2006年1月第1版第1次印刷

787mm × 1092mm 1/16 · 26.5印张

印数：0 001- 4 000册

定价：48.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

# 前　言

在目前正在使用的程序设计语言之中，汇编语言是被误解得最深的一种。当提到“汇编语言”这个术语时，经常使人联想到低级的位移动和在长达数千页的指令手册中费力地查找正确的指令格式。随着各种出色的高级语言开发工具的快速发展，在各种程序设计新闻组中“汇编语言程序设计已经死亡了”这种评论并不少见。

但是，汇编语言程序设计远没有到死亡的时候。每种高级语言程序在能够连接为可执行程序之前都必须被编译为汇编语言程序。对于高级语言程序设计者来说，了解编译器如何生成汇编语言代码很有用处，这表现在使用汇编语言直接编写例程和了解编译器如何把高级语言转换为汇编语言方面。

## 本书目的

本书的主要目的是向使用高级语言的程序员讲解高级语言程序是如何被转换为汇编语言的，以及如何掌握生成的汇编语言代码。这就是说，本书的主要读者是已经熟悉高级语言（比如C、C++，甚至Java）的程序员。本书没有花时间讲解基本的程序设计原则。我们假设读者已经熟悉计算机程序设计的基础，并且有兴趣学习汇编语言以便了解程序运行的幕后发生了什么。

但是，如果你是程序设计的初学者并且把汇编语言程序设计作为起点，本书也没有完全忽略你的要求。可以从头到尾阅读各个章节获得如何进行汇编语言程序设计（和一般的程序设计）的基础知识。书中的每个主题都包括范例代码来演示汇编语言指令如何工作。如果你完全是程序设计新手，也可以从本书开始学习程序设计，进而学习本书其他高级主题。

## 本书范围

本书的主要目的是使C和C++程序员熟悉汇编语言，讲解编译器如何从C和C++程序创建汇编语言例程，并讲解如何整理生成的汇编语言例程以便提高应用程序的性能。

所有用高级语言（比如C和C++）编写的程序，在被连接为可执行程序之前，都会被编译器转换为汇编语言。编译器使用编译器的设计者定义的特定规则来确定如何正确地转换高级语言语句。很多程序员只是编写高级语言程序并且假设编译器会创建正确的可执行代码来实现程序。

但是，情况并非总是如此。当编译器把高级语言代码转换为汇编语言代码时，稀奇古怪的事情经常出现。另外，编译器往往遵循非常特别的转换规则，以至于不能在最终的汇编语言代码中发现节省时间的捷径，而对于编写不良的高级例程，它也不能加以改善。在这样的情况下，汇编语言代码的知识就有用武之地了。

本书正是讲解汇编语言的知识，描述在连接为可执行程序之前如何检查编译器生成的汇编

---

参与本书翻译工作的有：马朝晖、迟旭、陈美红、楼涵、翁彩霞、郑纪革、何运刚、张志刚、李晓东、曾明月、刘磊。

语言代码，并发现可以修改何处代码来提高性能或者提供附加功能，帮助读者理解编译器的转换处理是如何影响高级语言例程的。

## 本书结构

本书分为三个部分。第一部分讲解汇编语言程序设计环境的基础。因为汇编语言在各种处理器和汇编器之间是不同的，所以必须选择常见的平台。本书使用运行在Intel处理器系列上的Linux操作系统。Linux环境提供丰富的程序开发工具，比如优化编译器、汇编器、连接器和调试器，它们的费用很低，或者是免费的。Linux环境之中这些丰富的开发工具使它非常适合把C程序剖析为汇编语言代码。

第一部分的各章如下：

第1章“什么是汇编语言”，一开始确保你确切地了解什么是汇编语言以及如何将它融入程序设计模型。这一章揭开了汇编语言的神秘面纱，并且提供了解如何把汇编语言和高级语言一起使用的基础知识。

第2章“IA-32平台”，提供对Intel奔腾处理器系列的简要介绍。当使用汇编语言时，了解底层的处理器和它如何处理程序是很重要的。但是这一章没有打算对IA-32平台的操作进行深入的分析，也没有提供在这个平台上进行程序设计涉及到的硬件和操作。

第3章“相关的工具”，讲解本书中使用的Linux开放源代码的开发工具。本书中使用GNU编译器、汇编器、连接器和调试器对程序进行编译、汇编、连接和调试。

第4章“汇编语言程序范例”，演示如何在Linux系统上使用GNU工具创建、汇编、连接和调试简单的汇编语言程序。这一章还演示如何在Linux系统上在汇编语言程序中使用C库函数为汇编语言应用程序添加额外的特性。

本书的第二部分研究汇编语言程序设计的基础。在能够分析编译器生成的汇编语言代码之前，必须了解汇编语言指令。这一部分的各章如下：

第5章“传送数据”，讲解在汇编语言程序中如何传送数据元素。讲解寄存器、内存位置和堆栈的概念，并且提供在它们之间传送数据的范例。

第6章“控制执行流程”，描述汇编语言程序中使用的分支指令。这可能是程序最为重要的特性之一，认识分支并且优化分支的能力对提高应用程序的性能是至关重要的。

第7章“使用数字”，讨论在汇编语言中如何使用不同的数字数据类型。能够在汇编语言程序内正确地处理整数和浮点值是很重要的。

第8章“基本数学功能”，讲解如何使用汇编语言指令实现基本的数学功能，比如加、减、乘和除。虽然这些通常是直接的功能，但是可以使用灵活的技巧提高这一领域工作的性能。

第9章“高级数学功能”，讨论IA-32浮点运算单元（Floating Point Unit，FPU），以及如何使用它处理复杂的浮点运算。浮点运算对于数据处理程序经常是至关重要的，了解它如何工作对高级语言程序员有非常大的好处。

第10章“处理字符串”，讲解汇编语言的各种字符串处理指令。字符数据是高级语言程序设计的另一个重要方面。在高级语言中处理字符串时，了解汇编语言层面上如何处理字符串能够提供深入的认识。

第11章“使用函数”，开始深入地讲解汇编语言程序设计。创建汇编语言函数去执行例程是汇编语言优化的核心。了解汇编语言函数的基础知识是有益的，因为编译器从高级语言代码生成汇编语言代码时经常会使用它们。

第12章“使用Linux系统调用”，这一章结束了这一部分，它演示使用已经创建的函数可以在汇编语言中执行多少高级功能。Linux系统提供很多高级功能，例如输出到显示器的功能。在汇编语言程序中经常可以利用它们。

第三部分讲解更加高级的汇编语言主题。因为本书的主要主题是如何在C或者C++代码之中并入汇编语言例程，所以最初的几章只是讲解如何这样做。其余的各章讲解一些更加高级的主题，圆满地完成读者对汇编语言程序设计的学习。这一部分包括下面这几章：

第13章“使用内联汇编”，讲解如何把汇编语言例程直接并入你的C或者C++语言程序中。内联汇编语言经常用于在C程序中“硬编码”快速例程，以便确保编译器为例程生成适当的汇编语言代码。

第14章“调用汇编库”，演示可以如何把汇编语言函数组合为库，供众多应用程序（包括汇编语言程序和高级语言程序）使用。能够把频繁使用的函数组合为C或者C++程序可以调用的单一库是非常节省时间的特性。

第15章“优化例程”，本书的核心部分：修改编译器生成的汇编语言代码以便满足特定要求。这一章讲解在汇编语言代码中究竟如何生成不同类型的C例程（比如if-then语句和for-next循环）。了解了汇编语言代码在做什么之后，就可以对它进行修改以便为特定的环境定制代码。

第16章“使用文件”，讲解汇编语言程序设计中最被忽视的功能之一。几乎所有应用程序都需要对系统进行某种类型的文件访问。汇编语言程序也不例外。这一章讲解如何使用Linux的文件处理系统调用读取、写入和修改系统中文件内的数据。

第17章“使用高级IA-32特性”，这一章结束了本书，它讲解高级的Intel的单指令多数据（Single Instruction Multiple Data，SIMD）技术。这种技术为程序员提供在单一指令之中执行多个运算操作的平台。在音频和视频数据处理的领域之中，这种技术变得非常重要。

## 使用本书的要求

本书中的所有范例都用汇编语言编写，并且运行在Linux操作系统和Intel处理器平台上。本书中广泛地使用开放源代码的GNU编译器（gcc）、汇编器（gas）、连接器（ld）和调试器（gdb）演示汇编语言特性。第4章专门讨论如何在Linux平台上使用这些工具创建、汇编、连接和调试汇编语言程序。如果读者没有安装Linux平台，第4章演示了如何使用可以直接从光盘引导的、无需修改工作站硬盘的Linux版本。无需在工作站上安装Linux，就可以使用本书中使用的所有GNU开发工具。

## 约定

为了帮助读者从书中得到最大的收益以及充分理解内容，我们在本书中使用了数种约定。

技巧、提示、诀窍和当前讨论之外的内容使用楷体排版。在代码范例中我们使用灰色背景突出显示新的和重要的代码，而当前上下文中不那么重要的代码以及以前显示过的代码将不用

灰色背景。

## 源代码

当你研究本书中的范例时，可以选择手工输入所有代码，或者从网上下载本书源代码文件。本书中用到的所有示例源代码都可以从www.wrox.com下载。访问这个站点时，只需查找本书的名称（可以使用Search文本框，也可以使用书名列表之一），然后在本书的具体介绍页面上点击Download Code链接即可获得本书的所有源代码。[⊖]

因为很多书籍的名称类似，所以查找书籍最简单的方式是通过ISBN查找；本书的ISBN是0-7645-7901-0。

下载代码后可以使用常用的压缩工具解压。或者可以到Wrox公司的代码下载页面（www.wrox.com/dynamic/books/download.aspx）查看本书和所有其他Wrox的书籍的代码。

## 勘误

我们尽了最大的努力来确保文本或者代码中没有错误。但是，没有人能够做到完美，错误在所难免。如果发现我们的书中有错误，比如拼写错误或者有错误的代码段，我们将非常高兴收到反馈。通过发送勘误表，可能避免另外一个读者数小时的迷惑，同时帮助我们提供品质更好的信息。

要查看本书的勘误表，可以访问www.wrox.com并且使用Search文本框或者书名列表之一查找书籍名称。然后，在书籍的具体介绍页面上点击Book Errata链接。在这个页面上，可以查看关于本书已经提交的和Wrox公司的编辑公布的所有勘误。完整的书籍列表，包括到每本书的勘误的链接，都可以在www.wrox.com/misc-pages/booklist.shtml页面上找到。

如果没有在Book Errata页面上找到“你的”错误，请访问www.wrox.com/ contact/techsupport.shtml并且完整填写这里的表单，把发现的错误发送给我们。我们会查看这些信息，并且，如果正确的话，会在书籍的勘误页面上公布消息并在书籍以后的版本中修正这些问题。

## p2p.wrox.com

为了与作者和同行们进行讨论，请加入P2P论坛：p2p:wrox.com。这个论坛是基于Web的系统，你可以张贴关于Wrox公司书籍的和关于技术的消息，可以和其他读者以及技术用户进行交流。论坛提供订阅功能，当论坛上发布新帖子的时候，可以通过电子邮件发送你所选择的感兴趣的主题。Wrox公司的作者、编辑、其他业界专家以及本书读者都会出现在这些论坛上。

在http://p2p.wrox.com上，你会发现许多不同的论坛，在你阅读本书及开发自己的应用程序时，它们都能够给你提供帮助。可以按照下面的步骤加入论坛：

1) 访问p2p.wrox.com，点击Register链接。

2) 阅读使用条款并且点击Agree。

3) 填写必需的信息以及你希望提供的任何可选的信息，点击Submit。

---

⊖ 也可登录华章网站（www.hzbook.com）下载源代码。

4) 然后你会收到描述如何验证帐户和完成加入过程的电子邮件。

不加入P2P也可以阅读论坛的消息，但是要想发布自己的消息，就必须加入。

加入后，你可以发布新的消息并且回复其他用户发布的消息。任何时候都可以在Web上阅读消息。如果希望特定的论坛使用电子邮件通知新的消息，可以在论坛列表中点击这个论坛的Subscribe链接。

关于如何使用Wrox公司的P2P的更多信息，请阅读P2P FAQ，它们回答了关于论坛软件如何工作的问题，这些FAQ也包括关于P2P和Wrox公司书籍的很多常见问题。要阅读FAQ，可以在任何P2P页面上点击FAQ链接。

# 目 录


前言

## 第一部分 汇编语言程序设计环境基础

第1章 什么是汇编语言 ……1
1.1 处理器指令 ……1
1.1.1 指令码处理 ……1
1.1.2 指令码格式 ……2
1.2 高级语言 ……5
1.2.1 高级语言的种类 ……5
1.2.2 高级语言的特性 ……7
1.3 汇编语言 ……8
1.3.1 操作码助记符 ……8
1.3.2 定义数据 ……9
1.3.3 命令 ……11
1.4 小结 ……11
第2章 IA-32平台 ……13
2.1 IA-32处理器的核心部分 ……13
2.1.1 控制单元 ……14
2.1.2 执行单元 ……18
2.1.3 寄存器 ……19
2.1.4 标志 ……21
2.2 IA-32的高级特性 ……23
2.2.1 x87浮点单元 ……23
2.2.2 多媒体扩展 ……24
2.2.3 流化SIMD扩展 ……24
2.2.4 超线程 ……25
2.3 IA-32处理器系列 ……25
2.3.1 Intel处理器 ……25
2.3.2 非Intel处理器 ……26
2.4 小结 ……27
第3章 相关的工具 ……29
3.1 开发工具 ……29
3.1.1 汇编器 ……29
3.1.2 连接器 ……31
3.1.3 调试器 ……31
3.1.4 编译器 ……32
3.1.5 目标代码反汇编器 ……32
3.1.6 简档器 ……33
3.2 GNU汇编器 ……33
3.2.1 安装汇编器 ……33
3.2.2 使用汇编器 ……35
3.2.3 关于操作码语法 ……36
3.3 GNU连接器 ……37
3.4 GNU编译器 ……39
3.4.1 下载和安装gcc ……39
3.4.2 使用gcc ……40
3.5 GNU调试器程序 ……42
3.5.1 下载和安装gdb ……42
3.5.2 使用gdb ……42
3.6 KDE调试器 ……44
3.6.1 下载和安装kdbg ……44
3.6.2 使用kdbg ……45
3.7 GNU objdump程序 ……46
3.7.1 使用objdump ……46
3.7.2 objdump范例 ……47
3.8 GNU简档器程序 ……48
3.8.1 使用简档器 ……48
3.8.2 简档范例 ……50
3.9 完整的Linux开发系统 ……51
3.9.1 Linux基础 ……51
3.9.2 下载和运行MEPIS ……52
3.9.3 新的开发系统 ……53
3.10 小结 ……53

第4章 汇编语言程序范例 ……55
4.1 程序的组成 ……55
4.1.1 定义段 ……55
4.1.2 定义起始点 ……55
4.2 创建简单程序 ……56
4.2.1 CPUID指令 ……56
4.2.2 范例程序 ……58
4.2.3 构建可执行程序 ……60
4.2.4 运行可执行程序 ……60
4.2.5 使用编译器进行汇编 ……60
4.3 调试程序 ……61
4.4 在汇编语言中使用C库函数 ……65
4.4.1 使用printf ……66
4.4.2 连接C库函数 ……67
4.5 小结 ……68

## 第二部分 汇编语言程序设计基础

第5章 传送数据 ……71
5.1 定义数据元素 ……71
5.1.1 数据段 ……71
5.1.2 定义静态符号 ……73
5.1.3 bss段 ……73
5.2 传送数据元素 ……75
5.2.1 MOV指令格式 ……75
5.2.2 把立即数传送到寄存器和内存 ……76
5.2.3 在寄存器之间传送数据 ……77
5.2.4 在内存和寄存器之间传送数据 ……77
5.3 条件传送指令 ……83
5.3.1 CMOV指令 ……83
5.3.2 使用CMOV指令 ……85
5.4 交换数据 ……86
5.4.1 数据交换指令 ……87
5.4.2 使用数据交换指令 ……91
5.5 堆栈 ……93
5.5.1 堆栈如何工作 ……93
5.5.2 压入和弹出数据 ……94
5.5.3 压入和弹出所有寄存器 ……96
5.5.4 手动使用ESP和EBP寄存器 ……97
5.6 优化内存访问 ……97
5.7 小结 ……98
第6章 控制执行流程 ……99
6.1 指令指针 ……99
6.2 无条件分支 ……100
6.2.1 跳转 ……100
6.2.2 调用 ……103
6.2.3 中断 ……106
6.3 条件分支 ……106
6.3.1 条件跳转指令 ……106
6.3.2 比较指令 ……108
6.3.3 使用标志位的范例 ……109
6.4 循环 ……112
6.4.1 循环指令 ……112
6.4.2 循环范例 ……113
6.4.3 防止LOOP灾难 ……113
6.5 模仿高级条件分支 ……114
6.5.1 if语句 ……115
6.5.2 for循环 ……118
6.6 优化分支指令 ……120
6.6.1 分支预测 ……120
6.6.2 优化技巧 ……122
6.7 小结 ……124
第7章 使用数字 ……126
7.1 数字数据类型 ……126
7.2 整数 ……127
7.2.1 标准整数长度 ……127
7.2.2 无符号整数 ……128
7.2.3 带符号整数 ……129
7.2.4 使用带符号整数 ……131
7.2.5 扩展整数 ……131
7.2.6 在GNU汇编器中定义整数 ……134
7.3 SIMD整数 ……135
7.3.1 MMX整数 ……136
7.3.2 传送MMX整数 ……136
7.3.3 SSE整数 ……137

7.3.4 传送SSE整数 ……138
7.4 二进制编码的十进制 ……139
7.4.1 BCD是什么 ……140
7.4.2 FPU BCD值 ……140
7.4.3 传送BCD值 ……141
7.5 浮点数 ……142
7.5.1 浮点数是什么 ……143
7.5.2 标准浮点数据类型 ……144
7.5.3 IA-32浮点值 ……146
7.5.4 在GNU汇编器中定义浮点值 ……146
7.5.5 传送浮点值 ……146
7.5.6 使用预置的浮点值 ……148
7.5.7 SSE浮点数据类型 ……149
7.5.8 传送SSE浮点值 ……150
7.6 转换 ……153
7.6.1 转换指令 ……154
7.6.2 转换范例 ……154
7.7 小结 ……155
第8章 基本数学功能 ……157
8.1 整数运算 ……157
8.1.1 加法 ……157
8.1.2 减法 ……165
8.1.3 递增和递减 ……169
8.1.4 乘法 ……169
8.1.5 除法 ……173
8.2 移位指令 ……175
8.2.1 移位乘法 ……175
8.2.2 移位除法 ……177
8.2.3 循环移位 ……178
8.3 十进制运算 ……178
8.3.1 不打包BCD的运算 ……178
8.3.2 打包BCD的运算 ……180
8.4 逻辑操作 ……181
8.4.1 布尔逻辑 ……182
8.4.2 位测试 ……182
8.5 小结 ……183
第9章 高级数学功能 ……185
9.1 FPU环境 ……185
9.1.1 FPU寄存器堆栈 ……185
9.1.2 FPU状态、控制和标记寄存器 ……185
9.1.3 使用FPU堆栈 ……190
9.2 基本浮点运算 ……193
9.3 高级浮点运算 ……196
9.3.1 浮点功能 ……196
9.3.2 部分余数 ……199
9.3.3 三角函数 ……201
9.3.4 对数函数 ……203
9.4 浮点条件分支 ……205
9.4.1 FCOM指令系列 ……205
9.4.2 FCOMI指令系列 ……207
9.4.3 FCMOV指令系列 ……208
9.5 保存和恢复FPU状态 ……209
9.5.1 保存和恢复FPU环境 ……209
9.5.2 保存和恢复FPU状态 ……210
9.6 等待和非等待指令 ……213
9.7 优化浮点运算 ……213
9.8 小结 ……214
第10章 处理字符串 ……216
10.1 传送字符串 ……216
10.1.1 MOVS指令 ……216
10.1.2 REP前缀 ……220
10.1.3 其他REP指令 ……224
10.2 存储和加载字符串 ……225
10.2.1 LODS指令 ……225
10.2.2 STOS指令 ……225
10.2.3 构建自己的字符串函数 ……226
10.3 比较字符串 ……227
10.3.1 CMPS指令 ……228
10.3.2 CMPS和REP一起使用 ……229
10.3.3 字符串不等 ……230
10.4 扫描字符串 ……232
10.4.1 SCAS指令 ……232
10.4.2 搜索多个字符 ……233

10.4.3 计算字符串长度 ……235
10.5 小结 ……236
第11章 使用函数 ……237
11.1 定义函数 ……237
11.2 汇编函数 ……238
11.2.1 编写函数 ……239
11.2.2 访问函数 ……240
11.2.3 函数的放置 ……242
11.2.4 使用寄存器 ……242
11.2.5 使用全局数据 ……243
11.3 按照C样式传递数据值 ……244
11.3.1 回顾堆栈 ……244
11.3.2 在堆栈之中传递函数参数 ……244
11.3.3 函数开头和结尾 ……246
11.3.4 定义局部函数数据 ……246
11.3.5 清空堆栈 ……247
11.3.6 范例 ……248
11.3.7 在操作之中监视堆栈 ……249
11.4 使用独立的函数文件 ……252
11.4.1 创建独立的函数文件 ……252
11.4.2 创建可执行文件 ……253
11.4.3 调试独立的函数文件 ……254
11.5 使用命令行参数 ……255
11.5.1 程序剖析 ……255
11.5.2 分析堆栈 ……255
11.5.3 查看命令行参数 ……257
11.5.4 查看环境变量 ……258
11.5.5 使用命令行参数的范例 ……259
11.6 小结 ……261
第12章 使用Linux系统调用 ……262
12.1 Linux内核 ……262
12.1.1 内核组成 ……262
12.1.2 Linux内核版本 ……267
12.2 系统调用 ……268
12.2.1 查找系统调用 ……268
12.2.2 查找系统调用定义 ……269
12.2.3 常用系统调用 ……270
12.3 使用系统调用 ……271
12.4 复杂的系统调用返回值 ……275
12.4.1 sysinfo系统调用 ……276
12.4.2 使用返回结构 ……277
12.4.3 查看结果 ……278
12.5 跟踪系统调用 ……278
12.5.1 strace程序 ……278
12.5.2 高级strace参数 ……279
12.5.3 监视程序系统调用 ……280
12.5.4 附加到正在运行的程序 ……282
12.6 系统调用和C库 ……284
12.6.1 C库 ……284
12.6.2 跟踪C函数 ……285
12.6.3 系统调用和C库的比较 ……287
12.7 小结 ……287

## 第三部分 高级汇编语言技术

第13章 使用内联汇编 ……289
13.1 什么是内联汇编 ……289
13.2 基本的内联汇编代码 ……292
13.2.1 asm格式 ……292
13.2.2 使用全局C变量 ……294
13.2.3 使用volatile修饰符 ……296
13.2.4 使用替换的关键字 ……296
13.3 扩展asm ……296
13.3.1 扩展asm格式 ……296
13.3.2 指定输入值和输出值 ……297
13.3.3 使用寄存器 ……298
13.3.4 使用占位符 ……299
13.3.5 引用占位符 ……301
13.3.6 替换的占位符 ……302
13.3.7 改动的寄存器列表 ……303
13.3.8 使用内存位置 ……304
13.3.9 使用浮点值 ……305
13.3.10 处理跳转 ……306
13.4 使用内联汇编代码 ……308
13.4.1 什么是宏 ……308

13.4.2 C宏函数 ……309
13.4.3 创建内联汇编宏函数 ……310
13.5 小结 ……311
第14章 调用汇编库 ……312
14.1 创建汇编函数 ……312
14.2 编译C和汇编程序 ……313
14.2.1 编译汇编源代码文件 ……314
14.2.2 使用汇编目标代码文件 ……314
14.2.3 可执行文件 ……315
14.3 在C程序中使用汇编函数 ……317
14.3.1 使用整数返回值 ……317
14.3.2 使用字符串返回值 ……318
14.3.3 使用浮点返回值 ……321
14.3.4 使用多个输入值 ……322
14.3.5 使用混合数据类型的输入值 ……323
14.4 在C++程序中使用汇编函数 ……327
14.5 创建静态库 ……328
14.5.1 什么是静态库 ……328
14.5.2 ar命令 ……328
14.5.3 创建静态库文件 ……329
14.5.4 编译静态库 ……331
14.6 使用共享库 ……331
14.6.1 什么是共享库 ……331
14.6.2 创建共享库 ……332
14.6.3 编译共享库 ……332
14.6.4 运行使用共享库的程序 ……333
14.7 调试汇编函数 ……334
14.7.1 调试C程序 ……334
14.7.2 调试汇编函数 ……336
14.8 小结 ……337
第15章 优化例程 ……338
15.1 优化编译器代码 ……338
15.1.1 编译器优化级别1 ……338
15.1.2 编译器优化级别2 ……339
15.1.3 编译器优化级别3 ……341
15.2 创建优化的代码 ……341
15.2.1 生成汇编语言代码 ……341
15.2.2 查看优化的代码 ……344
15.2.3 重新编译优化的代码 ……345
15.3 优化技巧 ……345
15.3.1 优化运算 ……345
15.3.2 优化变量 ……348
15.3.3 优化循环 ……352
15.3.4 优化条件分支 ……356
15.3.5 通用子表达式消除 ……361
15.4 小结 ……363
第16章 使用文件 ……365
16.1 文件处理顺序 ……365
16.2 打开和关闭文件 ……366
16.2.1 访问类型 ……366
16.2.2 UNIX权限 ……367
16.2.3 打开文件代码 ……368
16.2.4 打开错误返回代码 ……369
16.2.5 关闭文件 ……370
16.3 写入文件 ……370
16.3.1 简单的写入范例 ……370
16.3.2 改变文件访问模式 ……372
16.3.3 处理文件错误 ……372
16.4 读取文件 ……373
16.4.1 简单的读取范例 ……374
16.4.2 更加复杂的读取范例 ……375
16.5 读取、处理和写入数据 ……377
16.6 内存映射文件 ……379
16.6.1 什么是内存映射文件 ……379
16.6.2 mmap系统调用 ……380
16.6.3 mmap汇编语言格式 ……381
16.6.4 mmap范例 ……383
16.7 小结 ……387
第17章 使用高级IA-32特性 ……388
17.1 SIMD简介 ……388
17.1.1 MMX ……388
17.1.2 SSE ……389
17.1.3 SSE2 ……389
17.2 检测支持的SIMD操作 ……390

17.2.1 检测支持 ……390
17.2.2 SIMD特性程序 ……391
17.3 使用MMX指令 ……392
17.3.1 加载和获得打包的整数值 ……393
17.3.2 执行MMX操作 ……393
17.4 使用SSE指令 ……400
17.4.1 传送数据 ……400
17.4.2 处理数据 ……401
17.5 使用SSE2指令 ……405
17.5.1 传送数据 ……405
17.5.2 处理数据 ……406
17.6 SSE3指令 ……408
17.7 小结 ……409

# 第一部分 汇编语言程序设计环境基础

## 第1章 什么是汇编语言

学习汇编语言的首要是了解什么是汇编语言。与其他程序设计语言不同的是，并不是所有汇编器都使用一种标准格式。不同的汇编器使用不同的语法编写程序语句。在试图掌握无数种汇编语言程序设计时，很多汇编语言程序设计的初学者都遇到了困难。

学习汇编语言程序设计的第一个步骤是，决定在现有环境中希望（或者需要）使用什么类型的汇编语言。一旦决定了要使用的汇编语言，开始学习并且在独立的和高级语言的程序中使用它就容易了。

本章首先讲解汇编语言是从哪里发展而来的，并且说明为什么要使用汇编语言进行程序设计。要了解汇编语言程序设计，必须首先了解它的最基本目的——使用处理器指令代码进行程序设计。接下来，这一章讲解编译器和连接器如何把高级语言转换为原始指令代码。学习了这些知识之后，就容易了解汇编语言程序和高级语言程序的区别在哪里，还有如何同时使用它们以相互补充。

### 1.1 处理器指令

在操作的最低层，所有计算机处理器（微型计算机、小型计算机和大型计算机）都按照制造厂商在处理器芯片内部定义的二进制代码操作数据。这些代码定义处理器应该利用程序员提供的数据完成什么功能。这些预置的代码被称为*指令码*（instruction code）。不同类型的处理器包含不同类型的指令码。通常按照处理器芯片支持的指令码的数量和类型对它们进行分类。

虽然不同类型的处理器可能包含不同类型的指令码，但是它们处理指令码程序的方式是类似的。这一小节讲解处理器如何处理指令，以及一个范例处理器芯片的指令码是什么样子。

#### 1.1.1 指令码处理

当计算机处理器芯片运行时，它读取存储在内存中的指令码。每个指令码集合可能包含一个或者多个字节的信息，这些信息指示处理器完成特定的任务。每条指令码都是从内存读取的，指令码所需的数据也是存储在内存中并且从内存读取。包含指令码的内存字节和包含处理器使用的数据的字节没有区别。

为了区分数据和指令码，要使用专门的*指针*（pointer）帮助处理器跟踪数据和指令码存储在内存中的什么位置。这显示在图1-1中。

*指令指针*（instruction pointer）用于帮助处理器了解哪些指令码已经处理过了，以及接下来要处理的是哪条指令码。当然，有些专门的指令能够改变指令指针的位置，比如跳转到程序的

特定位置。

类似的，数据指针（data pointer）用于帮助处理器了解内存中数据区域的起始位置是哪里。这个区域称为堆栈（stack）。当新的数据元素被放入堆栈中时，指针在内存中“向下”移动。当数据被读取出堆栈时，指针在内存中“向上”移动。

每条指令码都包含一个或者多个字节的处理器要处理的信息。例如，下面这些指令码字节（十六进制格式）

```
C7 45 FC 01 00 00 00
```

通知Intel IA-32系列处理器把十进制值1加载到一个处理器寄存器定义的内存偏移位置。指令码包含若干信息片段（在后面的“操作码”小节中定义），它们明确地定义处理器要完成什么操作。处理器完成了一个指令码集合的处理之后，它读取内存中的下一个指令码集合（就是指令指针所指向的）。在内存中，指令必须按照正确的格式和顺序放置，使处理器能够正确地按顺序执行程序代码。

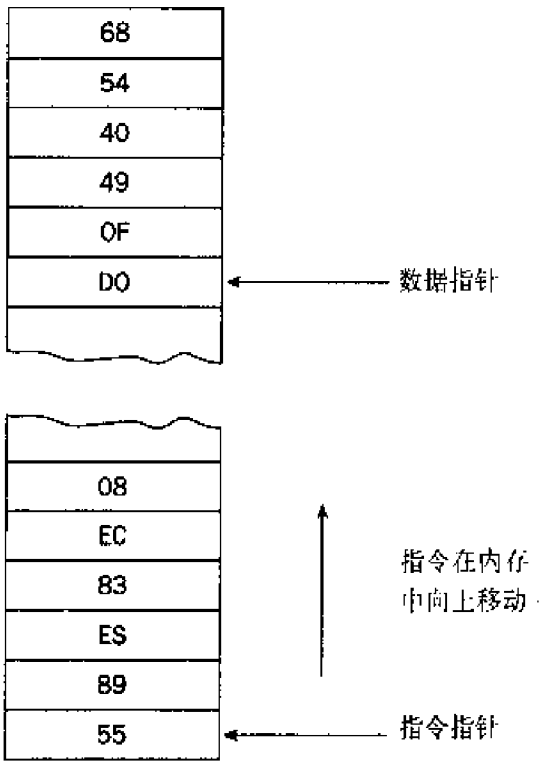



图 1-1

每条指令都必须至少包含1个字节的操作码（operation code，简写为opcode）。操作码定义处理器应该完成什么操作。每个处理器系列都具有其自己的预定义好的操作码，它们定义所有可用的功能。下一小节介绍Intel IA-32系列微处理器中使用的操作码是如何构成的。本书中的所有例子都使用这种类型处理器的操作码。

### 1.1.2 指令码格式

Intel IA-32系列微处理器包括现代IBM平台的微型计算机所使用的所有当前类型的微处理器（参见第2章），还包括流行的奔腾系列微处理器。IA-32系列的微处理器使用专门格式的指令码，了解这些指令的格式对汇编语言程序设计将有所帮助。IA-32指令码格式由四个主要部分构成：

- 可选的指令前缀
- 操作码（opcode）
- 可选的修饰符
- 可选的数据元素

图1-2是IA-32指令码格式的布局。

每个部分都用于完整地为处理器定义要执行的特定指令。下面的几个小节介绍指令码的每个部分和它们如何定义由处理器执行的指令。

Intel的奔腾处理器系列不是使用IA-32指令码格式的唯一处理器芯片系列。AMD公司生产的一系列芯片也完全兼容Intel IA-32指令码格式。

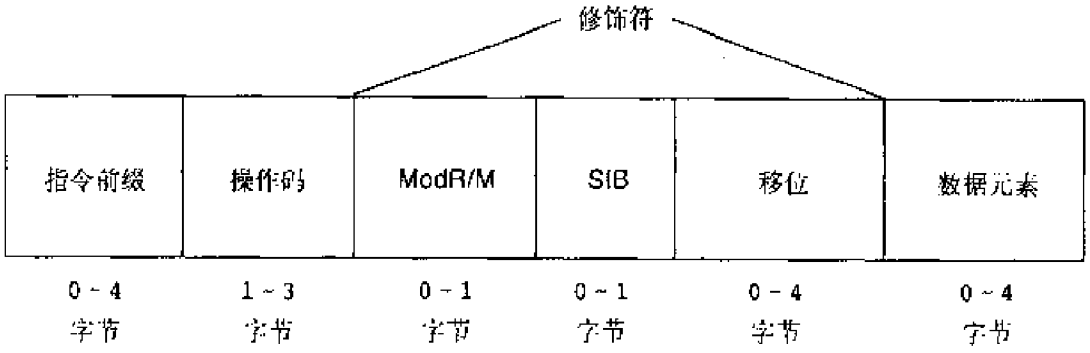



图 1-2

1. 操作码

如图1-2所示，IA-32指令码格式中唯一必须的部分是操作码。每个指令码都必须包含操作码，它定义由处理器执行的基本功能或者任务。

操作码的长度在1到3字节之间，它唯一地定义要执行的功能。例如，2字节的操作码OF A2定义IA-32 CPUID指令。当处理器执行这个指令码时，它返回不同寄存器中关于微处理器的特定信息。然后，程序员可以使用其他的指令码从处理器寄存器中提取信息，以便确定运行程序的微处理器的类型和型号。

寄存器是处理器芯片之内的组件，用于临时存储处理器正在处理的数据。寄存器的详细信息将在第2章“IA-32平台”中讲解。

2. 指令前缀

指令前缀可以包含1个到4个修改操作码行为的1字节前缀。按照前缀的功能，这些前缀被分为4个组。修改操作码时，每个组的前缀一次只能使用一个（因此最多有4个前缀字节）。这4个前缀组如下：

- 锁定前缀和重复前缀
- 段覆盖前缀和分支提示前缀
- 操作数长度覆盖前缀
- 地址长度覆盖前缀

锁定前缀表示指令将独占地使用共享内存区域。这对于多处理器和超线程系统非常重要。重复前缀用于表示重复的功能（常常在处理字符串时使用）。

段覆盖前缀定义可以覆盖定义了的段寄存器值的指令（这将在第2章中更加详细地讲解）。分支提示前缀尝试向处理器提供程序在条件跳转语句中最可能的路径的线索（这同预报分支的硬件一起使用）。

操作数长度覆盖前缀通知处理器，程序将在这个操作码之内切换16位和32位的操作数长度。这使程序可以在使用大长度的操作数时警告处理器，帮助加快对寄存器的数据赋值。

地址长度覆盖前缀通知处理器，程序将切换16位和32位的内存地址。这两种长度都可以被声明为程序的默认长度，这个前缀通知处理器程序将切换到另一种长度。

3. 修饰符

一些操作码需要另外的修饰符来定义执行的功能中涉及到什么寄存器和内存位置。修饰符

包含在3个单独的值中：

- 寻址方式说明符（ModR/M）字节
- 比例-索引-基址（SIB）字节
- 1、2或者4个的地址移位字节

(1) ModR/M字节

ModR/M字节由3个字段的信息构成，如图1-3所示。

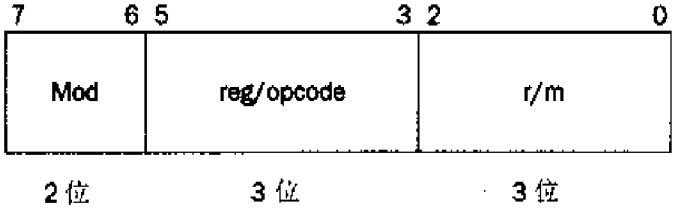



图 1-3

mod字段和r/m字段一起使用，用于定义指令中使用的寄存器或者寻址模式。在指令之中，可能的寻址模式有24个，加上8个可以使用的通用寄存器，所以有32个可能值。

reg/opcode字段用于允许使用更多的3位进一步定义操作码功能（比如操作码子功能），或者可以用于定义寄存器值。

r/m字段用于定义用作该功能的操作数的另一个寄存器，或者可以把它和mod字段组合在一起定义指令的寻址模式。

(2) SIB字节

SIB字节也由3个字段的信息构成，如图1-4所示。

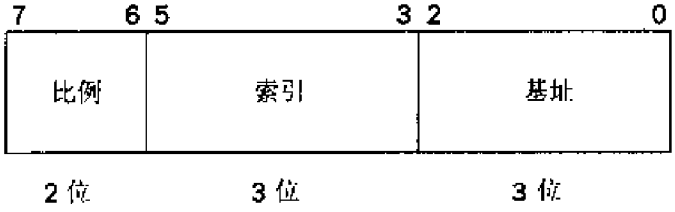



图 1-4

比例字段指定操作的比例因子。索引字段指定内存访问中用作索引寄存器的寄存器。基址字段指定用作内存访问的基址寄存器的寄存器。

ModR/M和SIB字节的组合创建一个表，它可以定义用于访问数据的众多可能的寄存器组合和内存模式。Intel奔腾处理器的规范说明书定义了ModR/M和SIB字节可以使用的所有可能的组合。

(3) 地址移位字节

地址移位字节用来指定对于ModR/M和SIB字节中定义的内存位置的偏移量。可以使用它作为基本内存位置的索引，用于存储或者访问内存之内的数据。

4. 数据元素

指令码的最后一部分是该功能使用的数据元素。一些指令码从内存位置或者处理器寄存器读取数据，而一些指令码在其本身之内包含数据。这个值经常被用于表示静态数字值（比如要

加的数字）或者内存位置。根据数据长度，这个值可以包含1、2或者4字节的信息。

例如，下面是前面显示过的指令码的例子：

```
C7 45 FC 01 00 00 00
```

它定义操作码C7，这个操作码是把值传送到内存位置的指令。内存位置由修饰符45 FC定义（它定义从EBP寄存器中的值（值45）指向的内存位置开始的4字节（值FC））。最后4字节定义放到这个内存位置的整数值（在这个例子中这个值是1）。

从这个例子可以看出，值1被写为4字节的十六进制值01 00 00 00。数据流中的字节的顺序取决于使用的处理器的类型。IA-32平台处理器使用“小尾数（little-endian）”表示法，其中低值的字节首先出现（当从左到右读时）。其他处理器使用“大尾数（big-endian）”顺序，其中高值字节首先出现。在汇编语言程序中指定数据和内存位置值时，这一概念非常重要。

## 1.2 高级语言

看上去使用纯粹的处理器指令码进行程序设计是困难的，确实是这样。即使最简单的程序也需要程序员指定许多操作码和数据字节。试图管理全都是指令码的大型程序是令人畏缩的任务。为了使程序员不致发狂，高级语言（high-level language，HLL）被创建出来了。

HLL使程序员可以使用简单的术语创建功能，而不是使用原始的处理器指令码。特殊的保留关键字用于定义变量（数据的内存位置）、创建循环（在指令码中进行跳转）和处理程序的输入和输出。但是，处理器根本不知道如何处理HLL代码。必须通过某种机制把代码转换为处理器能够处理的简单的指令码格式。这一小节定义HLL的不同类型，然后讲解如何把HLL代码转换为处理器可以执行的指令码。

### 1.2.1 高级语言的种类

程序员可以从很多不同的HLL之中作出选择，按照它们如何在计算机上运行，所有HLL可以分为两个不同的种类：

- 编译语言
- 解释语言

相同的程序设计语言的不同实现可能是编译的，也可能是解释的，这些种类用于说明特定的HLL实现如何定义怎么在处理器上运行程序。下面的章节讲解用于运行程序的方法和它们如何影响处理器操作它们的方式。

1. 编译语言

大多数产品应用程序都是使用编译HLL创建的。程序员使用语言的一般语句表达应用程序逻辑，以此创建程序。然后，文本程序语句被转换为可以在处理器上运行的指令码集合。通常，一般所说的编译一个程序实际上是两个步骤的过程：

- 把HLL语句编译为原始指令码
- 连接原始指令码来生成可执行程序

图1-5演示这一过程。

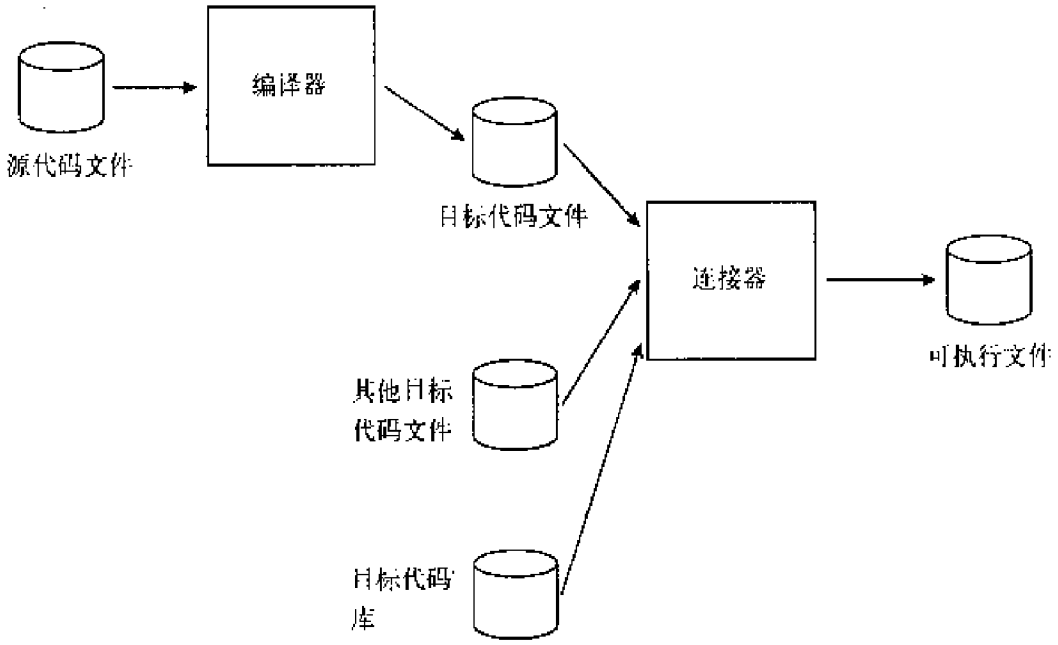



图 1-5

编译步骤把文本程序设计语言语句转换为实现应用程序功能所必须的指令码。每个HLL代码行都和要运行应用程序的特定处理器相关的一个或者多个指令码相匹配。例如，简单的HLL代码

```
int main()
{
    int i = 1;
    exit(0);
}
```

被编译为下面的IA-32指令码：

```
55
89 E5
83 EC 08
C7 45 FC 01 00 00 00
83 EC 0C
6A 00
E8 D1 FE FF FF
```

这个步骤生成一个中间文件，称为目标代码文件（object code file）。目标代码文件包含表示应用程序功能核心的指令码，如上所示。目标代码文件本身不能由操作系统运行。通常，宿主操作系统需要可执行文件（可以运行在系统上面的程序文件）的特殊文件格式，并且HLL程序可能需要其他目标文件的程序功能。所以需要另一个步骤来添加这些部分。

代码被编译为目标文件之后，要使用连接器（linker）把应用程序的目标代码文件和应用程序所需的所有附加的目标文件连接起来，并且创建最终的可执行输出文件。连接器输出的可执行文件只能够运行在编写程序所针对的操作系统上。不幸的是，每种操作系统使用的可执行文件的格式是不同的，所以在Microsoft Windows工作站上编译的应用程序不能在Linux工作站上工作，反过来也是一样。

包含常用功能的多个目标文件可以被组合为单一文件，称为库文件。然后，库文件可以被连接进多个应用程序，这可以在编译时进行（称为静态库），也可以在应用程序在系统上运行的

时候进行（称为动态库）。

2. 解释语言

编译程序自己运行在处理器上，而解释语言程序与之相反，它是由单独的程序读取和运行的。这个单独的程序是应用程序的宿主，它在进行处理时读取并且解释程序。程序运行时，把解释程序代码转换为适应处理器的正确指令码是宿主程序的任务。

显然，使用解释语言的缺陷是速度慢。程序没有直接被编译为运行在处理器上的指令码，而是由一个中间程序读取程序代码的每一行并且执行必须的功能。宿主程序读取代码并且执行代码所花费的时间给应用程序的执行增加了额外延迟。

既然使用解释语言会降低速度，读者可能会奇怪为什么还有人仍然要使用它们。一个回答是便利性。如果使用编译程序，每次对程序作出改动时，都必须重新编译程序并且和适当的代码库进行重新连接。如果使用解释程序，就可以对源代码文件进行快速的改动并且重新运行程序以便检查错误。还有，使用解释语言时，解释器应用程序自动地确定核心代码需要包含什么功能去支持程序的功能。

现在的程序设计语言环境模糊了编译语言和解释语言之间的界限。没有一种特定的语言可以被确定为属于某一个分类。相反，不同HLL的各个实现是可以分类的。例如，很多BASIC程序设计实现需要解释器把BASIC代码解释为可执行程序，但是也有很多BASIC实现允许程序员把BASIC程序编译为可执行指令码。

3. 混合语言

混合语言是程序设计的最新趋势，它把编译程序的特性和解释程序的通用性和简易性结合在了一起。混合语言的一个很好的例子就是流行的Java程序设计语言。

Java程序设计语言被编译为称为字节码（byte code）的形式。字节码和在处理器上看到的指令码类似，但是它本身不和当前的任何处理器系列兼容（虽然曾经有过设计能够把Java字节码当作指令集合运行的处理器的计划）。

相反，Java字节码必须通过Java虚拟机（Java Virtual Machine，JVM）进行解释，Java虚拟机单独运行在宿主计算机上。Java字节码是可移植的，就是说它可以通过任何类型的宿主计算机上的任何JVM运行。其优势在于不同的平台可以使用它们自己特定的JVM，这些JVM用于解释相同的Java字节码，而无需从原始的源代码进行重新编译。

### 1.2.2 高级语言的特性

如果是专业的程序员，那么很可能使用高级语言完成大部分（如果不是全部的话）编码工作。或许能够，或许不能有选择使用哪种HLL完成工程的机会，但是不管怎么样，HLL无疑使工作更加容易。这一小节讲解HLL最有用的两个特性：可移植性和标准化，它们是HLL和汇编语言程序设计的重要区别。

1. 可移植性

就像前面“处理器指令”一节中讲过的，指令码程序设计高度依赖计算机使用的处理器。每个不同的处理器系列都使用不同的指令码格式，存储数据的方法也不同（大尾数法和小尾数法）。为IA-32平台编写的指令码在MIPS处理器平台上面是不能工作的。

想像一下，为新的应用程序编写10 000行的指令码程序，它运行在Sun Sparc工作站上，然后要求把它移植到运行奔腾处理器的Linux工作站上。因为Sun Sparc工作站使用的微处理器使用的指令码和奔腾处理器不同，所以所有代码都必须重新编写为新的指令码——天啊!

通过简单地在新的平台上进行重新编译，HLL就能够移植到其他操作系统和其他处理器平台上。重新编译程序时，会自动地使用目标处理器的指令码重新编写它。

但是在实践中，使用了操作系统API的复杂程序要简单地重新编译为另一种平台的代码是困难的。例如，直接使用MS Windows的API的程序不能在Linux之下进行编译。

2. 标准化

HLL的另一个有用的特性是语言具有丰富的标准。电子及电气工程师协会（Institute of Electrical and Electronics Engineers，IEEE）和美国国家标准化组织（American National Standards Institute，ANSI）都为很多不同的HLL创建了标准规范。

这意味着获得了这样的保证：如果使用标准编译器在一种操作系统和处理器上编译了源代码，那么编译的结果和在另一种操作系统和处理器上编译的结果应该是相同的。每种编译器都被设计为把标准语言结构解释为目标处理器的指令码，以便跨处理器平台产生相同的功能。

## 1.3 汇编语言

虽然使用HLL创建大型应用程序通常比使用原始指令码要简单一些，但是这并不一定意味着产生的程序是有效率的。不幸的是，为了提高可移植性和符合标准，很多编译器编码为“最小公分母”。这就是说，为高级处理器芯片创建指令码的编译器可能不使用只有这些处理器才具有的特殊指令码，因此不能创建更快的应用程序。

市场上很多新的处理器提供的一个特性是高级的数学处理指令码。通过使用长度超常的字节表示数字（64位或者128位），这些指令码帮助提高处理复杂数学表达式的速度。不幸的是，很多编译器没有利用这些高级指令码的优势。幸运的是，程序员对此有简单的解决方案。在执行速度非常关键的环境之中，汇编语言程序设计可以提供帮助。当然，提高执行速度的首要步骤是确保使用最好的算法。对不良的算法进行优化是比不上使用更快的算法的。

汇编语言允许程序员直接创建指令码程序，而无需担心处理器上众多指令码集合的组合。相反，汇编语言程序使用助记符（mnemonics）表示指令码。助记符使程序员可以使用英语样式的词表示各个指令码。汇编器可以很容易地把汇编语言助记符转换为原始指令码。

本节讲解汇编语言助记符系统，以及如何使用它创建可以运行在处理器上的原始指令码程序。

汇编语言程序由3个组件构成，它们用于定义程序操作：

- 操作码助记符
- 数据段
- 命令

下面几节讲解每个组件和如何在汇编语言程序中使用它们创建最终的指令码程序。

### 1.3.1 操作码助记符

汇编语言程序的核心是用于创建程序的指令码。为了帮助轻松地编写指令码，汇编器把助

记符词汇和指令码功能（比如传送或者添加数据元素）等同对待。例如下面这个指令码的例子：

```
55
89 E5
83 EC 08
C7 45 FC 01 00 00 00
83 EC 0C
6A 00
E8 D1 FE FF FF
```

可以写为下面的汇编语言代码：

```
push %ebp
mov  %esp, %ebp
sub  $0x8, %esp
movl $0x1, -4(%ebp)
sub  $0xc, %esp
push $0x0
call 8048348
```

汇编语言程序员不必非要了解指令码的每个字节表示什么，他们可以使用更加容易记忆的助记符（比如push、mov、sub和call）来表示指令码。

不同的汇编器使用不同的助记符表示指令码。虽然出现了使汇编器助记符趋于标准化的趋势，但是助记符的种类还是非常繁多的，不仅在处理器系列之间是这样，在用于相同处理器指令码集合的汇编器之间也是如此。

每个处理器的生产厂商都发布详细讲解特定芯片集合执行的所有指令码的开发人员手册。Intel IA-32的开发人员手册可以在Intel的Web站点（www.intel.com）免费获得。这些开发人员手册超过1 000页，它们列举并且描述了奔腾处理器系列的所有指令码。

### 1.3.2 定义数据

除了指令码之外，大多数程序还需要使用数据元素保存程序当中用到的变量和常量值。HLL使用变量来定义保存数据的内存段。例如，在HLL程序中下面的代码是很常见的：

```
long testvalue = 150;
char message[22] = {"This is a test message"};
float pi = 3.14159;
```

这些语句的每一个都被HLL编译器理解为保留指定数量的字节的内存位置，这些位置用于存储程序执行期间可能改变，也可能不改变的值。程序每次引用变量名称（比如testvalue）的时候，编译器就知道要去访问内存中指定的位置来读取或者改变字节的值。

汇编语言也允许程序员定义将存储在内存中的数据项目。使用汇编语言进行程序设计的一个优势是它提供了更大的控制权，来决定在内存中的什么位置存储和如何存储数据。下面的小节讲解在汇编语言中用于存储和检索数据的两个方法。

1. 使用内存位置

和HLL定义数据的方法类似，汇编语言允许声明指向内存中特定位置的变量。在汇编语言中定义变量包括两个部分：

1) 指向一个内存位置的标记

2) 内存字节的数据类型和默认值

数据类型决定为变量保留多少字节。在汇编语言程序中就像下面这样:

```
testvalue:
   .long 150
message:
    .ascii "This is a test message"
pi:
   .float 3.14159
```

就像从数据类型看到的，汇编语言允许声明存储在内存位置中的数据的类型，还有放在内存位置的默认值，这和HLL的大多数方法类似。每种数据类型都占用特定数量的字节，从为标记保留的内存位置开始。如图1-6所示。

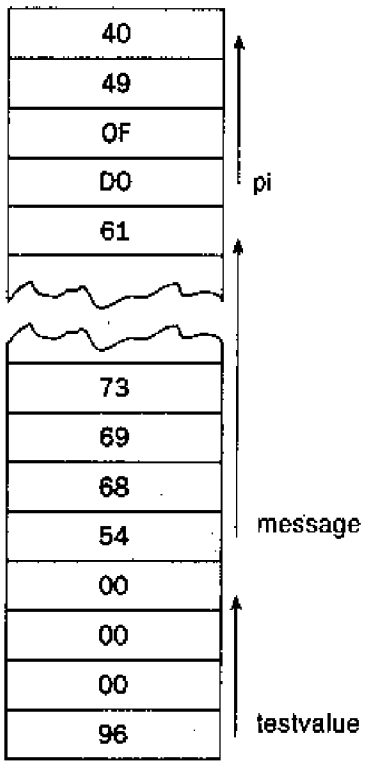



图 1-6

声明的第一个数据元素testvalue按照小尾数顺序作为4字节的十六进制值（96 00 00 00）被放在内存中。下一个数据元素message紧跟在数据元素testvalue的最后一个字节后面。因为数据元素message是文本值，所以它在内存中放置的顺序是按照字符串中出现的文本字符的顺序。最后，数据元素pi紧跟在数据元素message的最后一个字节后面（浮点数将在第7章“使用数字”中详细讨论）。

在汇编语言程序中按照定义开始位置的标记来引用内存位置。下面是汇编语言程序的一个例子:

```
movl testvalue, %ebx
addl $10, %ebx
movl %ebx, testvalue
```

第一条指令把testvalue标记指向的内存位置的4字节值（它的值被定义为150）载入EBX寄存器。下一条指令把EBX寄存器中存储的值加上10（十进制），然后把结果放回EBX寄存器。最后，寄存器的值被存储到testvalue标记引用的内存位置中。以后在程序中，可以再次使用testvalue标记引用这个新的值，它的值将是160（这一过程将在第5章和第8章中详细讲解）。

2. 使用堆栈

在汇编语言中用于存储和检索数据的另一种方法称为堆栈（stack）。堆栈是特殊的内存区域，经常保留作在程序中的函数之间传递数据元素。也可以使用它临时地存储和检索数据元素。

堆栈是在计算机为应用程序保留的内存范围的结尾位置保留的内存区域。一个指针（称为堆栈指针（stack pointer））用于指向堆栈中的下一个内存位置以便放入或者取出数据。堆栈很像一叠纸，在把数据元素放到堆栈中时，它就成为可以从堆栈中删除的第一个项目（假设只能从这叠纸的上面把纸拿走）。

在汇编语言程序中调用函数时，常常把希望传递给函数的任何数据元素放到堆栈的顶端。函数被调用时，它可以从堆栈查找数据元素。

存储和检索数据的不同方法将在第5章“传送数据”中详细讲解。

### 1.3.3 命令

指令和数据不是构成汇编语言程序的仅有的元素。汇编器保留专门的关键字用于在助记符被转换为指令码时，指示汇编器如何执行专门的函数。

在前面的小节中在定义数据元素时，已经见过了命令的例子。数据类型是使用GNU汇编器中使用的汇编器命令声明的。.long、.ascii和.float命令用于通知汇编器正在声明一个特定的数据类型。就像前面的例子显示的，命令前面有一个点号，这是命令和标记的不同之处。

命令是不同汇编器之间有区别的另一个方面。很多不同的命令用于帮助使程序员创建指令码的工作更加容易。一些现代的汇编器具有足以和很多HLL特性（比如while循环和if-then语句）相匹敌的命令清单！但是，老式的、更加传统的汇编器保持最少的命令，这迫使汇编语言程序员使用助记代码来创建程序逻辑。

汇编语言程序中使用的最为重要的命令之一是.section命令。这个命令定义内存段，汇编语言程序在其中定义元素。所有汇编语言程序都至少具有3个必须声明的段落：

- 数据段
- bss段
- 文本段

数据段用于声明为程序存储数据元素的内存区域。在声明数据元素之后，这一段落不能扩展，并且它在整个程序中保持静态。

bss段也是静态的内存段。它包含用于以后在程序中声明的数据的缓冲区。这一段落的特殊之处是缓冲区内存区域是由0填充的。

文本段是内存中存储指令码的区域。同样，这一区域也是固定的，其中只包含汇编语言程序中声明的指令码。

汇编语言程序中使用的这些命令将在第4章“汇编语言程序范例”中演示。

## 1.4 小结

虽然汇编语言程序设计经常被归为单一的程序设计语言类别，但是实际上有多种多样的不同类型的汇编语言汇编器。在汇编出最终的程序时，每种汇编器用来表示指令码、数据和专门命令的格式都稍有不同。使用汇编语言进行程序设计的第一个步骤是决定你需要使用哪种汇编器，并且了解它使用什么格式。

使用汇编语言的目的是使编码工作尽可能地靠近原始处理器指令码。处理器识别的代码称为指令码。每个处理器系列都有其自己的定义处理器可以执行的功能的指令码集合。另外，每个处理器系列都使用特定的指令码格式。Intel IA-32处理器系列使用由4个部分构成的格式。操作码用于定义应该使用哪条处理器指令。可选的前缀可以用于修改指令的行为。可选的修饰符可以用于定义指令中使用什么寄存器或者内存位置。最后，可以包含可选的数据元素，它定义指令中使用的特定数据值。

尝试使用原始指令码创建大型程序并不是简单的任务。为了使应用程序能够运行，每个指令码都必须按照正确的顺序逐字节地被编程。为了不强迫程序员学习所有指令码，开发人员开

发出了高级语言，高级语言使程序员可以使用便捷方法创建程序，然后由编译器把程序转换为正确的指令码。高级语言使用简单的关键字和术语定义一个或者多个指令码。这使程序员能够把精力集中在应用程序的逻辑上面，而不必担心底层处理器指令码的细节问题。

使用高级语言的缺陷是程序员依赖编译器创建者把程序设计逻辑转换为处理器运行的指令码。生成的指令码是否就是实现程序设计逻辑的最有效率的方法是没有保证的。对于希望得到最高效率或者希望对处理器如何处理程序具有更大控制能力的程序员来说，汇编语言程序设计提供了另外一种选择。

汇编语言程序设计使程序员可以使用指令码进行编程，但是使用简单的助记术语引用这些指令码。这样就向程序员同时提供了高级语言的简易性和使用指令码提供的控制能力。

不幸的是，汇编语言汇编器没有被标准化，并且汇编语言有很多不同的格式。所有汇编器都包括3个元素：操作码助记符、数据元素和命令。操作码助记符用于编写程序设计逻辑，数据元素用于定义保存常量和变量数据元素的内存位置。命令是最有争议的汇编器元素之一。命令帮助程序员定义特定的功能，比如声明数据类型，以及在程序之内定义内存区域。一些汇编器把命令提高到了更高的水平，提供支持很多高级语言功能（比如高级数据管理和逻辑程序设计）的命令。

下一章讨论Intel IA-32处理器系列的具体安排。在开始在奔腾处理器系列上进行程序设计之前，了解硬件如何安排的非常重要。了解处理器如何处理数据使你能够更加有效率地进行编程工作，提高应用程序的速度。

# 第 2 章　IA-32平台

成功地进行汇编语言程序设计的一个关键是了解编程的环境。编程环境中最重要的部分是处理器。为了能够利用处理器的基本功能和高级功能，了解将运行程序的硬件平台是至关重要的。通常，使用汇编语言的全部重点就是在应用程序中利用处理器的低层特性。了解可以使用什么元素来帮助获得可能的最高执行速度，这意味着快速应用程序和慢速应用程序有天壤之别。

在编写本书的时候，目前在工作站和服务器中最为常用的处理器平台是Intel奔腾系列的处理器。为奔腾处理器设计的硬件和指令码集合经常被称为IA-32平台。

本章讲解构成Intel IA-32平台的硬件元素。本章的第一部分讲解IA-32处理器平台中的基本组件。然后讲解IA-32系列中较新的奔腾4处理器芯片的高级特性。最后，讨论IA-32平台内包含的不同处理器，讲解使用不同类型的处理器（包括Intel和其他厂商生产的）需要注意什么特性。

## 2.1　IA-32处理器的核心部分

虽然不同的处理器系列结合了不同的指令集合和功能，但是大多数处理器都使用了相同的核心组件集合。大多数介绍性的计算机科学课程讲授计算机的4个基本组件。图2-1显示这些核心组件的基本框图。

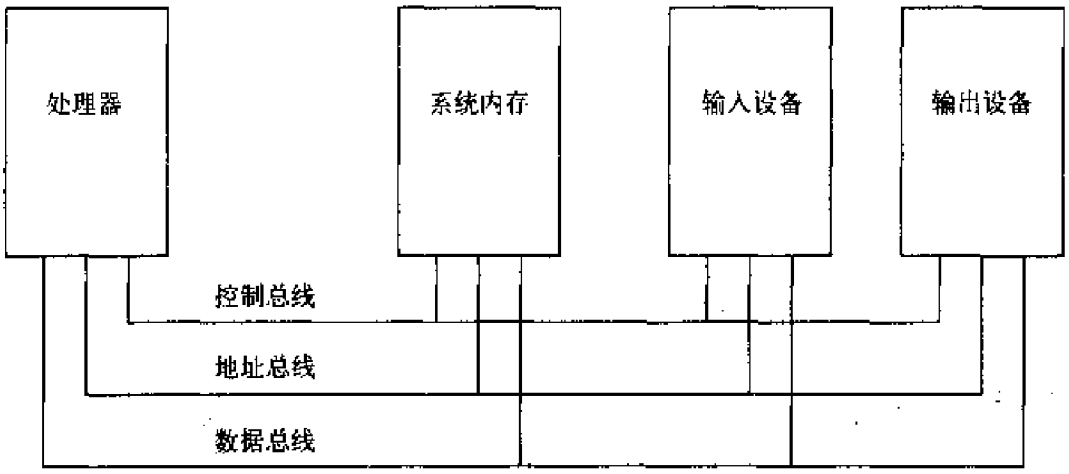



图 2-1

处理器包含控制计算机操作的硬件和指令码。通过使用3个单独的总线：控制总线、地址总线和数据总线，处理器被连接到计算机的其他元素（内存存储单元、输入设备和输出设备）。

控制总线用于保持处理器和各个系统元素之间功能的同步。数据总线用于在处理器和外部系统元素之间传送数据。这种操作的一个例子是从内存位置读取数据。处理器把要读取的内存地址放到地址总线上，然后内存存储单元作出响应，把这个内存位置存储的值放到数据总线上以便处理器进行访问。

处理器本身由很多组件构成。在处理器处理数据的过程中，每个组件都有其作用。汇编语

言程序具有访问和控制所有这些组件的能力，所以了解这些组件是什么是很重要的。处理器的主要组件如下:

- 控制单元
- 执行单元
- 寄存器
- 标志

图2-2显示这些组件以及它们如何在处理器内进行交互。

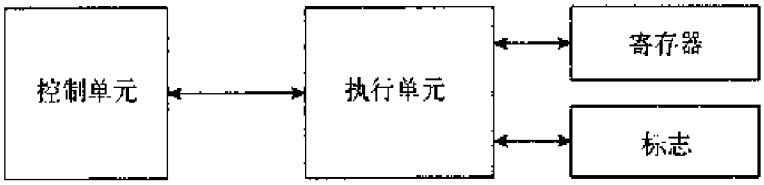



图 2-2

下面几节介绍每个核心组件，并且介绍它们在IA-32平台中如何实现。

### 2.1.1 控制单元

处理器的中心是控制单元。控制单元的主要作用是控制处理器内在任何时候进行什么操作。在处理器运行时，必须从内存获得指令并且加载它们以便处理器进行处理。控制单元的工作是实现4个基本功能:

1) 从内存获得指令。

2) 对指令进行解码以便进行操作。

3) 从内存获得所需的数据。

4) 如果必要，就存储结果。

指令计数器从内存获得下一条指令码并且使之准备好进行处理。指令解码器用于把获得的指令码解码为微操作。微操作是控制处理器芯片之内的特定信号来执行指令码的功能的代码。

微操作准备好之后，控制单元把它传递给执行单元进行处理，并且获得所有要存储在正确位置的结果。

控制单元是被研究得最多的处理器部分。在微处理器科技中很多发展都属于控制单元部分。在加快控制单元的操作方面，Intel作出了无数次的改进。其中最为有帮助的改进之一是控制单元获得和处理指令的方式。

在编写本书的时候，最新的Intel处理器（奔腾4）使用称为NetBurst的控制单元技术。NetBurst技术结合了4种单独的技术来帮助加快控制单元中的处理速度。了解这些技术如何起作用能够有助于优化汇编语言程序。NetBurst的特性如下:

- 指令预取和解码
- 分支预测
- 乱序执行
- 退役

这些技术一起工作，构成了奔腾4处理器的控制单元。图2-3显示这些元素如何进行交互。

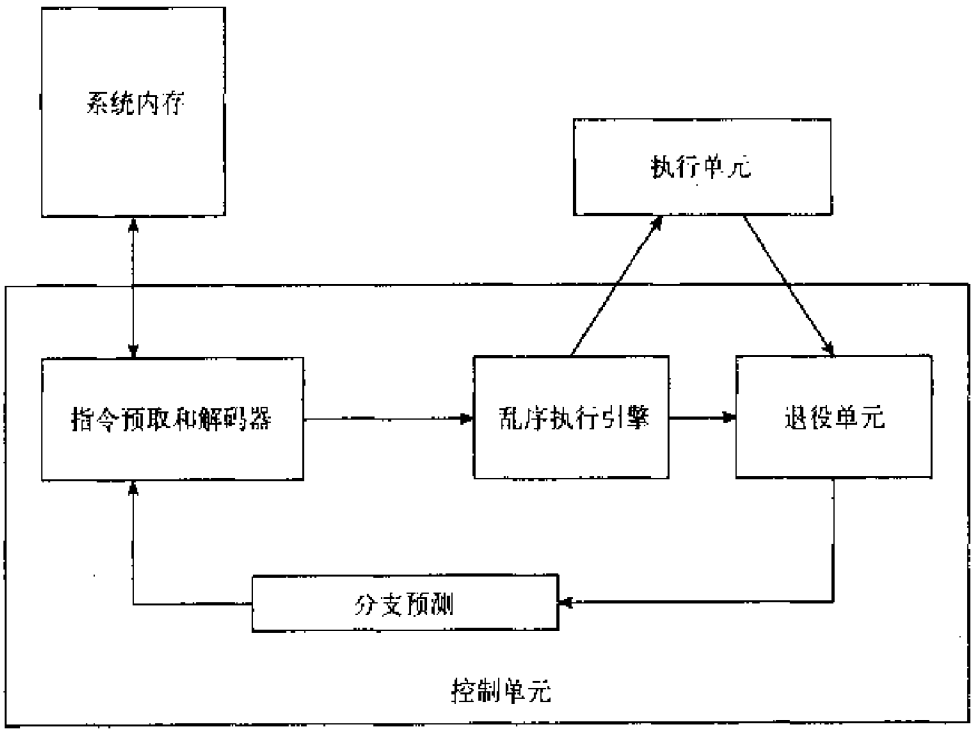



图 2-3

下面几节介绍所有这些技术在奔腾4处理器中是如何实现的。

1. 指令预取和解码管线

当执行单元需要指令和数据时，老式的IA-32系列处理器直接从系统内存获取它们。因为从内存获取数据的时间比处理它们的时间要长很多，所以就发生了待办工作积压的情况，因此处理器经常等待从内存获取指令和数据。为了解决这个问题，就建立了预取（prefetch）的概念。

虽然名称听上去有些古怪，但是预取涉及到试图在执行单元实际需要指令和/或数据之前获得（获取）它们。为了实现预取，处理器芯片本身需要一个专门的存储区域——处理器可以很容易地访问这一区域，比一般的内存访问要快。这是使用管线操作（pipeline）实现的。

管线操作涉及到在处理器芯片中创建内存缓存，在处理指令和数据元素之前，可以从缓存获取和存储它们。当执行单元为下一条指令作好准备时，这条指令已经在缓存中了并且可以被快速地处理。图2-4演示这个过程。

IA-32平台通过使用两级（或者更多级）的缓存实现管线操作。第一个缓存级别（称为L1）在它认为处理器将需要指令码和数据时，会试图从内存预取它们。随着指令指针在内存中移动，预取算法确定应该读取哪些指令码并且把它们放到缓存中。按照类似的方式，如果要处理内存中的数据，预取算法就会试图确定接下来可能要访问什么数据元素，也从内存读取它们并且把它们放在缓存中。

当然，对指令和数据进行缓存的一个缺陷在于不能保证程序将按照连续的顺序执行指令。如果程序采用某一逻辑分支，使指令指针转移到内存中完全不同的位置，那么整个缓存就没有用处了，必须清空整个缓存并且使用来自新位置的指令重新填充缓存。

为了缓解这一问题，就创建了第二个缓存级别。第二个缓存级别（称为L2）也可以保存指

令码和数据元素，它独立于第一个缓存级别。当程序逻辑跳转到内存中完全不同的区域去执行指令时，第二个级别的缓存仍然可以保存来自前面指令位置的指令。如果程序逻辑跳转回这个区域，那么这些指令仍然被缓存保存着，并且处理这些指令的速度几乎和存储在第一级缓存中的指令一样迅速。

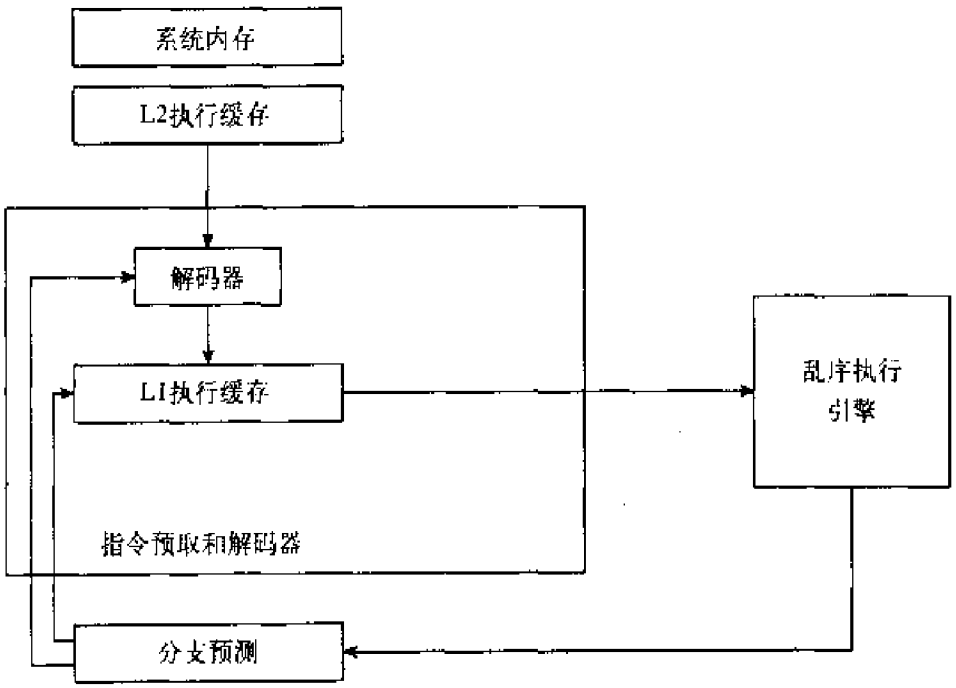



图 2-4

虽然汇编语言程序不能访问指令和数据缓存，但是了解这些元素如何工作也是有好处的。通过尽量减少程序中的分支，可以帮助提高程序中指令码的执行速度。

2. 分支预测单元

虽然实现多个级别的缓存是帮助加快程序逻辑的执行速度的一个途径，但是仍然没有解决“跳转的”程序的问题。如果程序采用很多不同的逻辑分支，那么使不同级别的缓存跟上分支的跳转也许是不可能的事情，结果就造成了更多在最后时刻对指令码和数据元素进行内存访问的情况。

为了解决这个问题，IA-32平台的处理器还引入了分支预测（branch prediction）。分支预测使用专门的算法试图预测接下来在程序分支中需要哪些指令码。

专门的统计学算法和分析被引入，用来确定指令码中最可能的执行路径。这条路径上的指令码被预取并且加载到缓存中。

奔腾4处理器使用3种技术实现分支预测：

- 深度分支预测
- 动态数据流分析
- 推理性执行

深度分支预测使处理器能够试图越过程序中的多个分支对指令进行解码。这里同样实现统计学算法来预测程序在分支中最可能的执行路径。虽然这种技术有所帮助，但是它并不是完全不出错误的。

动态数据流分析对处理器中的数据流进行统计学实时分析。被预测为程序流程必须经过的、

但是指令指针还没有达到的指令被传递给乱序执行引擎（下面讲解）。另外，在处理器等待与另一条指令相关的数据时，任何可以执行的指令都会被处理。

推理性执行使处理器能够确定指令码分支中不是立即就需要执行的哪些“远距离”指令码以后有可能需要执行，并且试图处理这些指令，这里同样使用乱序执行引擎。

3. 乱序执行引擎

在速度方面，乱序执行引擎是奔腾4处理器最了不起的改进之一。这里是为执行单元准备要处理的指令的地方。它包含几个缓冲区用于改变管线中指令的顺序，以便提高控制单元的性能。如图2-5所示。

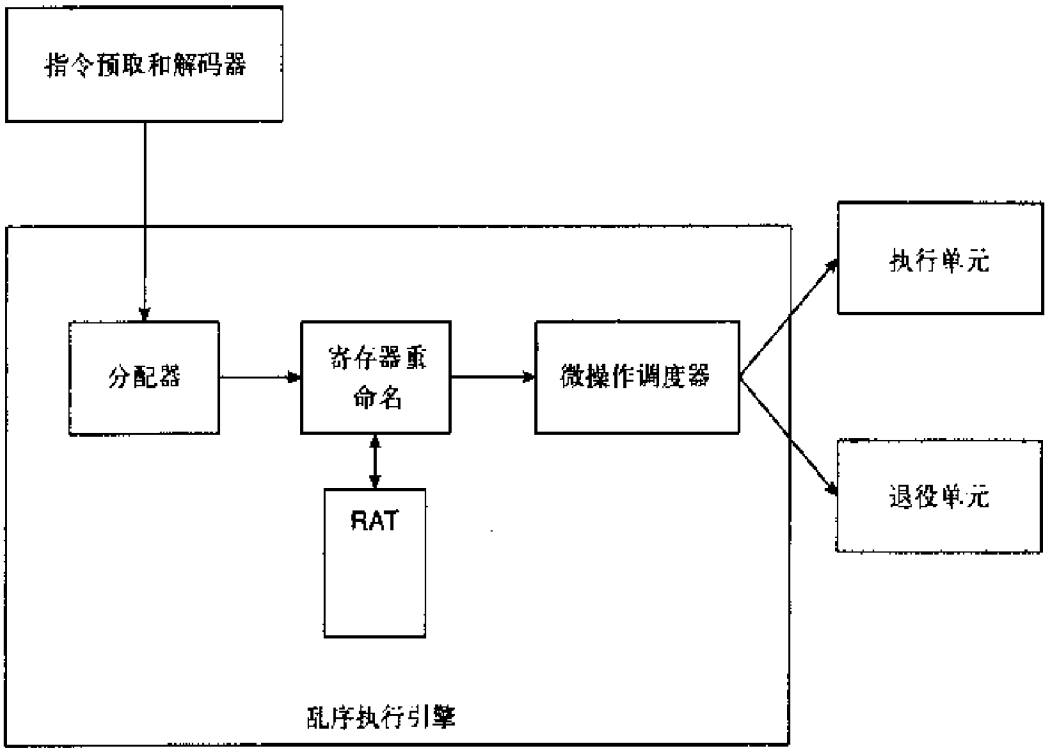



图 2-5

从预取和解码管线获取的指令被分析和重新排序，使它们能够尽快地被执行。通过分析大量指令，乱序执行引擎能够找到那些在程序的其他部分需要它们之前随时可以执行（并且保存它们的结果）的独立指令。在任何一个时刻，奔腾4处理器的乱序执行引擎中可以有最多126条指令。

乱序执行引擎内有3个部分：

- 分配器
- 寄存器重命名
- 微操作调度器

分配器是乱序执行引擎的交通警察。它的工作是确保缓冲区空间被适当地分配给乱序执行引擎正在处理的每条指令。如果所需的资源不可用，分配器会停止对指令的处理并且把资源分配给能够完成处理的另一条指令。

寄存器重命名部分分配逻辑寄存器去处理需要寄存器访问的指令。寄存器重命名部分不使用IA-32处理器上可用的8个通用寄存器（以后在“寄存器”小节中讲解），而是包含128个逻辑

寄存器。它把指令提出的寄存器请求映射到某个逻辑寄存器上，以便允许多条指令对相同寄存器同时进行访问。寄存器映射是使用寄存器分配表（register allocation table，RAT）完成的。这帮助提高需要访问相同寄存器集合的指令的处理速度。

通过检查微操作需要的输入元素，微操作调度器确定何时准备好处理微操作。它的工作是把准备好处理的微操作发送给退役单元，同时仍然维持程序相关性。微操作调度器使用两个队列，在它们中安排微操作——一个队列用于需要内存访问的微操作，一个用于不需要内存访问的。这两个队列被捆绑到分派端口。不同类型的奔腾处理器可能包含不同数量的分派端口。分派端口把微操作发送给退役单元。

4. 退役单元

退役单元接收从管线解码器和乱序执行引擎发送来的所有微操作，并且试图把微操作重新调整为程序能够正确执行的适当顺序。

退役单元按照乱序执行引擎发送微操作的顺序，把它们发送给执行单元进行处理，但是然后它监视结果，把结果重新调整为程序能够执行的正确顺序。

这些操作是使用一个大型缓冲区区域保存微操作的结果并且按照它们被需要的正确顺序放置它们来实现的。

当微操作完成并且结果被安排为正确的顺序之后，微操作就被认为是退役了，并且从退役单元删除。退役单元还更新分支预测单元中的信息，以便确保它知道哪些分支已经被采用了，以及哪些指令码已经被处理过了。

### 2.1.2 执行单元

处理器的主要功能是执行指令。这个功能是在执行单元中实现的。单一处理器实际上可以包含多个执行单元，能够同时处理多条指令码。

执行单元由一个或者多个运算逻辑单元（Arithmetic Logic Unit，ALU）构成。ALU被专门设计为处理不同数据类型的数学操作。奔腾4的执行单元包括用于下列功能的独立ALU:

- 简单整数操作
- 复杂整数操作
- 浮点操作

1. 低级潜在因素的整数执行单元

低级潜在因素的整数执行单元被设计为快速完成简单整数数学操作，比如加法、减法和布尔操作。奔腾4处理器在每个时钟周期能够完成两个低级潜在因素的整数操作，实际上使处理速度加倍了。

2. 复杂整数执行单元

复杂整数执行单元处理更加复杂的整数数学操作。复杂整数执行单元在4个时钟周期之内处理大多数移位和循环指令。乘法和除法操作涉及较长的计算时间，通常要花费14－60个时钟周期。

3. 浮点执行单元

在IA-32系列的不同处理器之间，浮点执行单元是有区别的。所有奔腾处理器都可以使用标准浮点执行单元处理浮点数学操作。包含MMX和SSE支持的奔腾处理器也在浮点执行单元中完

成这些计算。

浮点执行单元包含处理长度从64位到128位的数据元素的寄存器。这样就使在计算中使用更大的浮点值成为可能，这可以加快复杂浮点计算（比如数字信号处理和视频压缩）的速度。

### 2.1.3 寄存器

处理器的大多数操作都必须处理数据。不幸的是，处理器可能采取的最慢的操作是试图读取内存中的数据或者把数据存储到内存中。如图2-1所示，当处理器访问数据元素时，请求必须被发送到处理器外部、通过控制总线，然后进入内存存储单元。这一过程不仅复杂，而且在执行内存访问时迫使处理器处于等待状态。这一停机时间可以用于处理其他指令。

为了帮助解决这个问题，处理器包含内部的内存位置，称为寄存器（register）。寄存器能够存储要处理的数据元素，而无需访问内存存储单元。寄存器的不利之处在于处理器芯片中内置的寄存器数量是有限的。

IA-32平台具有不同长度的多组寄存器。IA-32平台中的不同处理器包含专门的寄存器。IA-32系列中所有处理器都可以使用的寄存器的核心组显示在下表中。

| 寄 存 器 | 描 述 |
|---|---|
| 通用 | 8个32位寄存器，用于存储正在处理的数据 |
| 段 | 6个16位寄存器，用于处理内存访问 |
| 指令指针 | 单一的32位寄存器，指向要执行的下一条指令码 |
| 浮点数据 | 8个80位寄存器，用于浮点数学数据 |
| 控制 | 5个32位寄存器，用于确定处理器的操作模式 |
| 调试 | 8个32位寄存器，用于在调试处理器时包含信息 |

下面几节更加深入地讲解比较常见的寄存器。

1. 通用寄存器

当处理器处理数据时，通用寄存器用于临时地存储数据。通用寄存器的应用从旧式的8位8080处理器的时代就开始了，一直发展到奔腾处理器中可用的32位寄存器。通用寄存器的每个新版本都被设计为完全向下兼容以前的处理器。因此，使用8080芯片上的8位寄存器的代码在32位的奔腾芯片上依然是有效的。

虽然大多数通用寄存器都可以用于存储任何类型的数据，但是其中一些具有专门的用途，它们在汇编语言程序中以一致的方式使用。下表列出了奔腾平台上可用的通用寄存器和它们最常被用于什么目的。

| 寄 存 器 | 描 述 |
|---|---|
| EAX | 用于操作数和结果数据的累加器 |
| EBX | 指向数据内存段中的数据的指针 |
| ECX | 字符串和循环操作的计数器 |
| EDX | I/O指针 |
| EDI | 用于字符串操作的目标的数据指针 |
| ESI | 用于字符串操作的源的数据指针 |
| ESP | 堆栈指针 |
| EBP | 堆栈数据指针 |

32位EAX、EBX、ECX和EDX寄存器也可用通过16位和8位名称引用，以表示寄存器的旧式版本。图2-6显示如何引用寄存器。

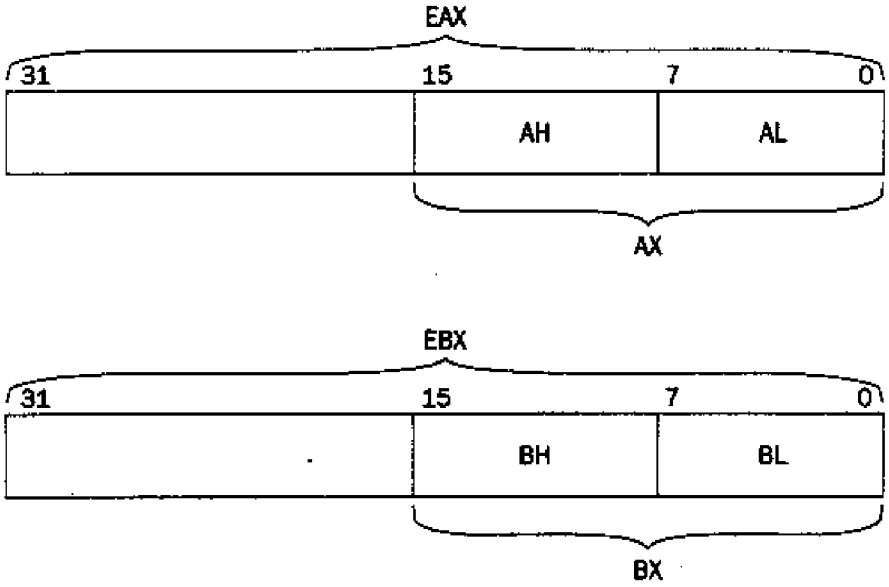



图 2-6

通过使用AX引用，EAX寄存器的低16位被使用。通过使用AL引用，EAX的低8位被使用。AH引用AL之后的高8位。

2. 段寄存器

段寄存器专门用于引用内存位置。IA-32处理器平台允许3种不同的访问系统内存的方法：

- 平坦内存模式
- 分段内存模式
- 实地址模式

平坦内存模式把全部系统内存表示为连续的地址空间。所有指令、数据和堆栈都包含在相同的地址空间中。通过称为线性地址（linear address）的特定地址访问每个内存位置。

分段内存模式把系统内存划分为独立段的组，通过位于段寄存器中的指针进行引用。每个段用于包含特定类型的数据。一个段用于包含指令码，另一个段用于包含数据元素，第三个段用于包含程序堆栈。

段中的内存位置是通过逻辑地址定义的。逻辑地址由段地址和偏移地址构成。处理器把逻辑地址转换为相应的线性地址位置以便访问内存的字节。

段寄存器用于包含特定数据访问的段地址。下表描述可用的段寄存器。

| 段寄存器 | 描　述 |
|---|---|
| CS | 代码段 |
| DS | 数据段 |
| SS | 堆栈段 |
| ES | 附加段指针 |
| FS | 附加段指针 |
| GS | 附加段指针 |

每个段寄存器都是16位的，包含指向内存特定段的起始位置的指针。CS寄存器包含指向内存中代码段的指针。代码段是内存中存储指令码的位置。处理器按照CS寄存器的值和EIP指令指针寄存器中包含的偏移值从内存获得指令码。程序不能显式地加载或者改变CS寄存器。当程序被分配一个内存空间时，处理器将为CS寄存器赋值。

DS、ES、FS和GS段寄存器都用于指向数据段。通过使用4个独立的数据段，程序可以分隔数据元素，确保它们不会重叠。程序必须加载带有段的正确指针值的数据段寄存器，并且使用偏移值引用各个内存位置。

SS段寄存器用于指向堆栈段。堆栈包含传递给程序中的函数和过程的数据值。

如果程序使用实地址模式，那么所有段寄存器都指向零线性地址，并且都不会被程序改动。所有指令码、数据元素和堆栈元素都是通过它们的线性地址直接访问的。

3. 指令指针寄存器

指令指针寄存器（即EIP寄存器），有时候称为程序计数器（program counter），它跟踪要执行的下一条指令码。虽然这听上去像一个简单的过程，但是对于指令预取缓存的实现却不是的。指令指针指向要执行的下一条指令。

应用程序不能直接修改指令指针本身。不能指定内存地址并且把它放到EIP寄存器中。相反，必须使用一般的程序控制指令（比如跳转）来改变要读入到预取缓存的下一条指令。

在平坦内存模式中，指令指针包含下一条指令码的内存位置的线性地址。如果应用程序使用分段内存模式，那么指令指针指向逻辑内存地址，通过CS寄存器的内容引用。

4. 控制寄存器

5个控制寄存器用于确定处理器的操作模式，还有当前正在执行的任务的特性。下表介绍各个控制寄存器。

| 控制寄存器 | 描　述 |
|---|---|
| CR0 | 控制操作模式和处理器状态的系统标志 |
| CR1 | 当前没有使用 |
| CR2 | 内存页面错误信息 |
| CR3 | 内存页面目录信息 |
| CR4 | 支持处理器特性和说明处理器特性能力的标志 |

不能直接访问控制寄存器中的值，但是可以把控制寄存器中包含的数据传送给通用寄存器。数据被传送给通用寄存器之后，应用程序就可以查看寄存器中的位标志以便确定处理器和/或当前正在运行的任务的操作状态。

如果必须改动控制寄存器的标志值，可以改动通用寄存器中的数据，然后把内容传送给控制寄存器。系统程序员经常修改控制寄存器中的值。一般的用户应用程序不经常修改控制寄存器项目，虽然它们可能经常查询标志值以便确定正在运行应用程序的宿主处理器芯片的能力。

### 2.1.4 标志

对于处理器中实现的每个操作，都必须有一种机制来确定操作是否成功。处理器标志用于实现这个功能。

标志对于汇编语言程序是重要的，因为它们是可以用于确定程序的功能是否成功执行的唯一途径。例如，如果应用程序执行减法操作，其结果为负值，处理器内一个专门的标志就会被设置。不检查这个标志，汇编语言程序就没办法了解是否发生了错误。

IA-32平台使用单一的32位寄存器包含一组状态、控制和系统标志。EFLAGS寄存器包含32位信息，这些信息被映射为表示信息的特定标志。一些位被保留供未来使用，允许在未来的处理器中定义附加的标志。编写本书时，有17位被用作标志。

按照功能，标志被分为3组：

- 状态标志
- 控制标志
- 系统标志

下面几节介绍每个组中的标志。

1. 状态标志

状态标志用于表明处理器进行的数学操作的结果。当前使用的状态标志显示在下表中。

| 标 志 | 位 | 名　称 |
|---|---|---|
| CF | 0 | 进位标志 |
| PF | 2 | 奇偶校验标志 |
| AF | 4 | 辅助进位标志 |
| ZF | 6 | 零标志 |
| SF | 7 | 符号标志 |
| OF | 11 | 溢出标志 |

如果无符号整数值的数学操作产生最高有效位的进位或者借位，进位标志就被设置为1。这表示数学操作中发生了寄存器溢出的情况。发生溢出时，寄存器中保存的数据不是数学操作的正确答案。

奇偶校验标志用于表明数学操作中的结果寄存器是否包含错误数据。作为简单的正确性检查，如果结果中为1的位的总数是偶数，奇偶校验标志就被设置为1；如果结果中为1的位的总数是奇数，它就被清零。通过检查奇偶校验标志，应用程序可以确定寄存器的数据是否被操作破坏了。

辅助进位标志用于二进制编码的十进制（Binary Coded Decimal，BCD）数学操作中（参见第7章“使用数字”）。如果用于运算的寄存器的第3位发生进位或者借位操作，辅助进位标志就被设置为1。

如果操作的结果为零，零标志就被设置为1。这经常用作确定数学操作的结果是否为零的便捷途径。

符号标志被设置为结果的最高有效位，这一位是符号位。它表明结果是正值还是负值。

当带符号整数运算的结果的正值过大，或者负值过小时，溢出标志用于正确地表示寄存器中的值。

2. 控制标志

控制标志用于控制处理器中的特定行为。当前，只定义了一个控制标志——DF标志，即方向标志。它用于控制处理器处理字符串的方式。

当DF标志被设置（置1）时，字符串指令自动递减内存地址以便到达字符串中的下一个字节。当DF标志被清零（置0）时，字符串指令自动递增内存地址以便到达字符串的下一个字节。

3. 系统标志

系统标志用于控制操作系统级别的操作。应用程序绝不应该试图修改系统标志。下表列出了系统标志。

| 标 志 | 位 | 名 称 |
|---|---|---|
| TF | 8 | 陷阱标志 |
| IF | 9 | 中断使能标志 |
| IOPL | 12和13 | I/O特权级别标志 |
| NT | 14 | 嵌套任务标志 |
| RF | 16 | 恢复标志 |
| VM | 17 | 虚拟8086模式标志 |
| AC | 18 | 对准检查标志 |
| VIF | 19 | 虚拟中断标志 |
| VIP | 20 | 虚拟中断挂起标志 |
| ID | 21 | 识别标志 |

陷阱标志被设置为1时启用单步模式。在单步模式中，处理器每次只执行一条指令，然后等待执行下一条指令的信号。在调试汇编语言程序时这一特性极为有用。

中断使能标志控制处理器如何响应从外部源接收到的信号。

I/O特权字段表明当前正在运行的任务的I/O特权级别。它定义I/O地址空间的访问级别。特权字段值必须小于或者等于访问I/O地址空间所必需的访问级别；否则，任何访问地址空间的请求都会被拒绝。

嵌套任务标志控制当前正在运行的任务是否链接到前一个执行的任务。它用于链接被中断和被调用的任务。

恢复标志控制处理器在调试模式中如何响应异常。

虚拟8086标志表明处理器在虚拟8086模式中进行操作，而不是保护模式或者实模式。

对准检查标志（和CR0控制寄存器中的AM位一起）用于启用内存引用的对准检查。

当处理器在虚拟模式中进行操作时，虚拟中断标志起IF标志的作用。

当处理器在虚拟模式中进行操作时，虚拟中断挂起标志用于表示一个中断正被挂起。

ID标志有意思的地方是它表示处理器是否支持CPUID指令。如果处理器能够设置或者清零这个标志，它就支持CPUID指令。如果不能，那么CPUID指令就不可用。

## 2.2 IA-32的高级特性

到目前为止讲到的IA-32平台的核心特性是这一系列的从80386开始的所有处理器都支持的。这一节讲解汇编语言程序员在创建专门为奔腾处理器设计的程序时可以利用的一些高级特性。

### 2.2.1 x87浮点单元

IA-32系列中的早期处理器必须用一个单独的处理器芯片执行浮点数学操作。80287和80387

处理器专门为计算机芯片提供浮点数学操作。需要快速地处理浮点操作的程序员被迫求助于附加的硬件来支持他们的需求。

从80486处理器开始，80287和80387芯片的高级运算功能被合并到了主处理器中。为了支持这些功能，需要附加的指令码以及附加的寄存器和执行单元。这些元素结合在一起被称为x87浮点单元（floating-point unit，FPU）。

x87 FPU引入了下列附加的寄存器：

| FPU寄存器 | 描　述 |
|---|---|
| 数据寄存器 | 用于浮点数据的8个80位寄存器 |
| 状态寄存器 | 报告FPU状态的16位寄存器 |
| 控制寄存器 | 控制FPU精度的16位寄存器 |
| 标记寄存器 | 描述8个数据寄存器的内容的16位寄存器 |
| FIP寄存器 | 指向下一条FPU指令的48位FPU指令指针（FPU instruction pointer，FIP） |
| FDP寄存器 | 指向内存中的数据的48位FPU数据指针（FPU data pointer，FDP） |
| 操作码寄存器 | 保存FPU处理的最后指令的11位寄存器 |

FPU寄存器和指令码使汇编语言程序能够快速地处理复杂的浮点数学功能，比如图形处理、数字信号处理和复杂业务应用程序所需的那些功能。与不带FPU的标准处理器中使用的软件模拟方式相比，FPU可以使处理浮点运算的速度有相当大的提高。只要有可能，汇编语言程序员就应该利用FPU处理浮点运算。

### 2.2.2 多媒体扩展

奔腾II处理器引入了另一种方法，程序员可以利用它实现复杂的整数算术运算操作。多媒体扩展（multimedia extension，MMX）是支持Intel的单指令多数据（Single Instruction，Multiple Data，SIMD）执行模型的第一种技术。

开发SIMD模型是为了处理多媒体应用程序中常见的较大的数字。SIMD模型使用扩展的寄存器长度和新的数字格式来加快实时多媒体表现所需的复杂数字处理。

MMX环境包含处理器可以处理的3种新的整数数据类型：

- 64位打包的字节整数
- 64位打包的字整数
- 64位打包的双字整数

第7章“使用数字”将详细地讲解这些数据类型。为了处理新的数据格式，MMX技术引入了8个FPU寄存器作为专用寄存器。MMX寄存器被命名为MM0到MM7，它们用于对64位打包的整数执行整数算术运算。

虽然MMX技术提高了复杂整数算术运算的处理速度，但是它对需要复杂浮点算术运算的程序没有任何帮助。这个问题是使用SSE环境解决的。

### 2.2.3 流化SIMD扩展

下一代SIMD技术是从奔腾III处理器开始实现的。流化SIMD扩展（Streaming SIMD

extension，SSE）增强了经常用于3D图形、动态视频和视频会议的复杂浮点算术运算的性能。

奔腾III处理器中SSE的第一个实现引入了8个新的128位寄存器（称为XMM0到XMM7）和一种新的数据类型——128位打包的单精度浮点数。SSE技术还引入了附加的新指令码，可以在单一指令中处理最多4个128位打包的单精度浮点数。

奔腾4处理器中SSE的第二个实现（SSE2）引入了SSE使用的相同的XMM寄存器，还引入了5种新的数据类型：

- 128位打包的双精度浮点数
- 128位打包的字节整数
- 128位打包的字整数
- 128位打包的双字整数
- 128位打包的四字整数

这些数据类型也在第7章中详细讲解。新的数据类型和相应的指令码使程序员能够在他们的程序中利用更加复杂的数学操作。128位双精度浮点数据类型允许用最短的处理器时间实现高级3D几何技术（比如光线跟踪）。

SSE的第三个实现（SSE3）没有创建任何新的数据类型，但是提供了几个新的指令用于处理XMM寄存器中的整数和浮点值。

### 2.2.4 超线程

奔腾4处理器系列添加的最令人兴奋的特性之一是超线程（hyperthreading）。超线程使单一IA-32处理器能够同时处理多个程序执行线程。

超线程技术由位于单一物理处理器中的两个或者多个逻辑处理器构成。每个逻辑处理器都包含通用、段、控制和调试寄存器的完整集合。所有逻辑处理器共享相同的执行单元。乱序执行引擎负责处理不同逻辑处理器提供的独立线程的指令码。

超线程的主要优势表现在操作系统级别。多任务操作系统（比如Microsoft Windows和各种UNIX实现）能够把应用程序线程分配给各个逻辑处理器。对于应用程序的程序员，超线程也许不会表现出这么大的好处。

## 2.3 IA-32处理器系列

编写本书时，IA-32系列处理器是桌面工作站和很多服务器环境中最为流行的计算机平台。使用IA-32平台的最流行的操作系统是Microsoft Windows，虽然IA-32平台上也运行其他流行的操作系统，比如Novell文件服务器，以及基于UNIX的操作系统，比如Linux和BSD衍生系统。

这些年以来，虽然IA-32处理器平台有了不少改进，但是所有IA-32处理器很多特性是通用的。本章讲到的特性构成了为IA-32平台编写的汇编语言程序的核心。但是，了解具体处理器可用的专门特性有助于很好地提高汇编语言程序编写的速度。这一小节讲解IA-32系列中可用的不同处理器，以及在该平台上进行程序设计时为什么必须要考虑到它们的特性。

### 2.3.1 Intel处理器

当然，Intel是IA-32平台处理器的主要供应商。在现在的计算环境中，最常使用的处理器平

台是奔腾处理器。遇到更早期的IA-32处理器（比如80486）硬件的情况是极为少见的。

不幸的是，几种不同类型的奔腾处理器仍然在商务、学校和家庭中使用。如果创建的汇编语言程序利用了最新的处理器才具有的高级IA-32特性，可能会限制应用程序的销路。相反，如果了解程序设计环境由某种特定类型的处理器构成，使用可用的高级特性会有助于应用程序在性能上击败竞争者。

本节介绍工作站和服务器上常用的不同类型的奔腾处理器，重点介绍每种处理器可用的特性。

1. 奔腾处理器系列

当然，奔腾处理器系列的核心是基本奔腾处理器。奔腾处理器是1993年发布的，作为80486处理器的替代品。奔腾处理器是第一款引入双执行管线的处理器，也是第一款使用完全32位地址总线和64位内部数据通路的处理器。

虽然奔腾处理器的性能优势是显而易见的，但是从程序设计的角度来说，奔腾处理器没有提供超越80486架构的任何新特性。它支持80486处理器的所有核心寄存器和指令码，包括内部FPU支持。

2. P6处理器系列

P6处理器系列是1995年用奔腾Pro处理器发布的。奔腾Pro处理器使用和最初的奔腾处理器不同的全新架构。P6系列处理器是第一种使用超标量微架构的处理器，它通过允许多执行单元和指令预取管线极大地提高了性能。

奔腾MMX和奔腾II处理器是P6系列的一部分，它们是使用MMX技术的第一批处理器，并且还引入了新的低功耗状态（允许在空闲时使处理器处于睡眠模式）。这一特性帮助降低功耗，并且成为便携型计算设备的理想平台。

奔腾III处理器是第一款使用SSE技术的处理器，允许程序员快速和轻松地执行复杂的浮点算术运算操作。

3. 奔腾4处理器系列

奔腾4处理器是2000年发布的，它再次造就了微处理器设计的新趋势。奔腾4使用Intel NetBurst架构，通过引入指令管线、乱序执行核心和执行单元，这一架构提供极快的处理速度。

奔腾4处理器支持SSE3——一种实现附加浮点操作以便支持高速多媒体计算的更加高级的SSE技术。

4. 奔腾Xeon处理器系列

在2001年，Intel发布了奔腾Xeon处理器。它的主要用途是多处理器服务器操作。它支持MMX、SSE、SSE2和SSE3技术。

### 2.3.2 非Intel处理器

虽然IA-32平台经常被认为是Intel的东西，但是市场上也有很多实现IA-32特性的其他非Intel处理器。汇编语言应用程序有可能运行在非Intel平台上面，所以了解平台之间的区别是很重要的。

1. AMD处理器

现在，Intel最大的竞争者是AMD。对应Intel奔腾处理器的每个版本，AMD都发布了与之竞争的处理器芯片。使用AMD处理器的Microsoft Windows工作站并不少见。下表列出了AMD处

理器的历史。

| AMD处理器 | 等同于 | 注 释 |
|---|---|---|
| K5 | 奔腾 | 100%软件兼容 |
| K6 | 奔腾MMX | 具有完全MMX支持的奔腾 |
| K6-2 | 奔腾II | 使用3D Now技术 |
| K6-III | 奔腾III | |
| Athlon | 奔腾4 | |
| Athlon XP | 奔腾4 w/SSE | |

对于汇编语言程序员来说，AMD和Intel处理器之间最重要的区别体现在使用SIMD技术时。虽然AMD复制了MMX技术，但是它没有完全复制更加新颖的SSE技术。当Intel在奔腾处理器II中引入SSE时，AMD在向另一个方向发展。AMD K6-2处理器使用不同的SIMD技术，称为3D Now。3D Now技术使用类似SSE的寄存器和数据类型，但是它不是软件兼容的。使用SSE功能进行程序设计时，这给谋求高速的程序员造成了非常大的困难。

在2001年发布Athlon XP处理器时，AMD支持SSE整数算术运算。编写本书时，最新的AMD处理器芯片完全支持SSE技术。

2. Cyrix处理器

虽然这几年Cyrix公司已经不再营运，但是它们的IA-32平台处理器仍然在很多工作站和低端服务器上使用。仍然有可能在各种环境中遇到Cyrix处理器。

Cyrix处理器系列的发展对应着Intel处理器的很多版本。Cyrix生产的第一款奔腾级别的处理器最初称为6x86处理器。它和奔腾处理器是100%软件兼容的。当Intel发布MMX技术时，Cyrix发布了6x86MX处理器（它们没有得到被称为MMX的许可，但是MX已经足够接近了）。

当Cyrix被卖给VIA芯片组公司时，原来的Cyrix处理器系列被重新命名。6x86处理器被称为M1，6x86MX处理器被称为M2。同样，这些处理器保持了和奔腾对应版本的兼容性。

在Cyrix处理器消逝之前，最后一个版本推向了市场。它称为Cyrix III，和奔腾III处理器兼容。不幸的是，和AMD类似，它也必须使用3D Now技术支持SSE，这使它与为SSE编写的汇编语言程序不兼容。

## 2.4 小结

在编写汇编语言程序之前，必须了解程序执行时使用的目标处理器。现在使用的最流行的处理器平台是Intel IA-32平台。这一平台包括Intel的奔腾系列处理器，以及AMD的Athlon处理器。

IA-32平台的旗舰产品是Intel 奔腾4处理器。它使用NetBurst架构快速和轻松地处理指令和数据。NetBurst架构的核心包括一个控制单元，一个执行单元、寄存器和标志。

控制单元控制执行单元如何处理指令和数据。通过在执行单元处理指令很久之前从内存中预取和解码指令达到相当高的处理速度。还可以不按顺序处理指令并且把结果存储起来，直到应用程序需要它们的时候。

奔腾4处理器的执行单元具有同时处理多条指令的能力。简单的整数处理被快速执行，并且被存储在控制单元的乱序区域中，直到程序需要它们为止。复杂的整数和浮点处理也被流水线

化以便提高性能。

寄存器用作处理器之内的本地数据存储区域，防止对数据进行代价巨大的内存访问。IA-32平台处理器提供几个通用寄存器，用于在程序执行时保存数据和指针。按照指令指针寄存器的值从内存获取指令。控制寄存器控制处理器的行为。

一个专门的寄存器保存确定处理器状态和操作的几个标志。每个标志代表处理器的一个不同操作。状态标志表明处理器执行的操作的结果。控制标志控制处理器如何进行特定操作。系统标志确定操作系统的行为，应用程序的程序员不应该修改这一标志。

IA-32平台上的创新不断出现而且效果显著。最近的处理器版本引入了很多新的特性。奔腾处理器引入了浮点单元（floating-point unit，FPU）来帮助进行浮点数学操作。

为了进一步支持复杂数学处理，单指令多数据（Single Instruction，Multiple Data，SIMD）技术允许处理整数和浮点数形式的大型数字值。多媒体扩展（Multimedia Extension，MMX）使程序员可以在高精度整数运算中使用64位整数。在这些改进之后，流化SIMD扩展（Streaming SIMD Extension，SSE）技术允许程序员使用128位单精度浮点值；接下来，SSE2技术允许使用128位双精度浮点数据值。这些新的数据类型显著提高了数学计算密集型程序的处理速度，比如用于多媒体处理和数字信号处理的程序。

使用IA-32平台进行程序设计时，要留意不同处理器的区别，并且要了解每种处理器类型支持的功能。IA-32平台的核心是最初的奔腾处理器。它支持核心IA-32寄存器和指令集，还有简单的内置FPU支持。和奔腾处理器类似，AMD发布了K5处理器，Cyrix发布了6x86处理器。这些处理器和IA-32指令码集合是100%软件兼容的。

Intel在奔腾II处理器系列中引入了MMX功能。AMD照着做，在K6处理器中引入了MMX特性，Cyrix也发布了6x86MX处理器。这些处理器都包含MMX寄存器，以及附加的MMX指令码。

SSE技术使IA-32领域变得复杂起来。奔腾II处理器引入了SSE寄存器和指令集，但是不幸的是，其他处理器生产厂商没有能够直接引入这些特性。作为替换，AMD和Cyrix在它们的K6-2（AMD）和Cyrix III（Cyrix）处理器中实现了3D Now技术。3D Now技术提供和SSE相同的64位整数数据类型，但是指令码是不同的。

编写本书时，奔腾4处理器是Intel的旗舰处理器产品。它支持SSE3技术，还有NetBurst架构。AMD以Athlon XP为竞争对手，它现在引入了SSE寄存器和指令集，使之与奔腾4处理器做到软件兼容。

既然已经了解了本书中使用的硬件平台，现在是研究软件开发环境的时候了。下一章讨论Linux操作系统环境中可用的汇编语言工具。通过使用Linux，你可以利用GNU开发工具，以最小的代价创建专业的软件开发环境。

# 第 3 章　相关的工具

既然已经熟悉了IA-32硬件平台，现在是研究创建汇编语言程序必需的软件的时候了。为了创建汇编语言程序，必须具有某种类型的开发环境。现有很多不同的汇编语言开发工具，商业的和免费的都有。必须决定哪种开发环境最适合你。

首先，本章讲解应该使用什么工具来创建汇编语言程序。其次讨论GNU项目提供的程序设计开发工具。每种工具都有介绍，包括它们的下载和安装。

## 3.1　开发工具

就像任何其他职业，程序设计也需要合适的工具来完成工作。为了创建良好的汇编语言开发环境，必须具有熟悉使用的适当的工具。在高级语言环境中，可以购买完整的开发环境，与之不同的是，通常必须组建汇编语言开发环境。最低限度应该有下面这些工具:

- 汇编器
- 连接器
- 调试器

还有，要为其他高级语言程序创建汇编语言例程，应该有下面这些工具:

- 高级语言的编译器
- 目标代码反汇编器
- 用于优化的简档生成工具

下面几节讲解每种工具，以及在汇编语言开发环境中如何使用它们。

### 3.1.1　汇编器

显然，为了创建汇编语言程序，需要一些工具把汇编语言源代码转换为处理器的指令码。这就是汇编器产生的原因。

就像第1章“什么是汇编语言”中讲到的，汇编器是为之进行程序设计的底层硬件平台特定的。每个处理器系列都有其自己的指令码集合。选择的汇编器必须能够生成所在系统（或者开发的系统）的处理器系列的指令码。

汇编器从程序员创建的源代码生成指令码。如果你还记得第1章的内容，那里讲过汇编语言源代码程序有3个部分:

- 操作码助记符
- 数据段
- 命令

不幸的是，每种汇编器对于每个部分使用的格式是不同的。使用一种汇编器进行程序设计可能和使用另一种汇编器完全不同。虽然基础是一样的，但是它们的实现有非常大的区别。

汇编器之间最大的区别是汇编器命令。操作码助记符和处理器指令码的关系密切，而各个汇编器命令对于各个汇编器是独特的。命令指示汇编器如何构建指令码程序。一些汇编器的指令数目是有限的，而另一些汇编器具有数量庞大的命令。命令做很多事情，从定义程序段到实现if-then语句或者while循环。

还必须考虑如何编写汇编语言程序。一些汇编器带有内置的编辑器，当键入代码时编辑器帮助识别不正确的语法；而其他汇编器只是简单的命令行程序，只能汇编现有的代码文本文件。如果选择的汇编器不包含编辑器，那么必须选择适合所处环境的良好的编辑器。虽然使用UNIX的vi编辑器能够处理简单的程序，但是读者可能不希望使用它编写10 000行的汇编语言程序。

选择汇编器的基本原则是：它有能力尽可能简单地为目标环境创建指令码程序。下面几节介绍Intel IA-32平台上一些常见的汇编器。

1. MASM

Intel平台上所有汇编器的鼻祖——Microsoft汇编器（Microsoft Assembler，MASM）是Microsoft公司的产品。它从IBM兼容PC的起始阶段就是出现了，它使程序员可以在DOS和Windows环境中生成汇编程序。

因为MASM存在了这么久，所以众多的指南、书籍和范例程序无处不在，其中很多是免费的或者费用很低的。虽然Microsoft不再把MASM作为独立产品销售，但是它仍然和Microsoft的编译器产品系列Visual Studio捆绑在一起。使用Visual Studio的好处在于它是功能完整的集成开发环境（Integrated Development Environment，IDE）。Microsoft还允许各个公司和组织免费分发MASM 6.0文件，使读者可以从命令提示行汇编程序。在Web上查找MASM 6.0会得到可以免费下载它的站点的清单。

除了MASM，一个独立的开发者Steve Hutchessen创建了MASM32开发环境。MASM32结合了原始的MASM汇编器和流行的Windows Win32应用编程接口（Application Programming Interface，API），主要在C和C++应用程序中使用。这使汇编语言程序员可以完全在汇编程序中创建成熟的Windows程序。MASM32的Web站点是www.masm32.com。

2. NASM

Netwide汇编器（Netwide Assembler，NASM）最初是为UNIX环境开发的商业汇编器包。最近开发者把NASM发布为UNIX和Microsoft环境下的开放源代码的软件。它和所有Intel指令码集合完全兼容，可以生成UNIX、16位MS-DOS和32位Microsoft Windows格式的可执行文件。

和MASM类似，有相当多的书籍和指南介绍NASM。NASM的下载页面是http://nasm.sourceforge.net。

3. GAS

免费软件基金会（Free Software Foundation，FSF）的GNU项目产生了很多在UNIX操作系统环境中使用的免费软件包。GNU汇编器（称为gas）是UNIX环境中可用的最为流行的跨平台汇编器。

说跨平台，确实如此。虽然前面提到汇编器是各个处理器系列特定的，但是gas是个例外。它被开发为可以在很多不同的处理器平台上操作。显然，它必须知道正在哪种平台上使用它，并且按照底层平台来创建指令码程序。通常，gas能够自动检测底层硬件平台并且创建适合此平台的正确的指令码，而无需操作者的干涉。

gas的一个独特特性是能够创建不同于程序设计所在平台的平台的指令码。这使程序员可以在基于Intel的计算机上工作，却为基于MIPS的系统创建汇编语言程序。当然，这样做的缺陷是程序员不能在宿主系统上测试生成的程序。

本书的所有范例都使用GNU汇编器。这不仅因为它是良好的独立汇编器，而且因为GNU C编译器也使用它把编译后的C和C++程序转换为指令码。了解了如何使用gas进行汇编语言程序设计，就可以很容易地在现有的C和C++应用程序中引入汇编语言功能，这是本书的重点之一。

4. HLA

高级汇编器（High Level Assembler，HLA）是Randall Hyde教授创建的。它在DOS、Windows和Linux操作系统上创建Intel指令码应用程序。

HLA的主要目的是向初级程序员讲授汇编语言。它引入很多高级命令帮助程序员从高级语言转向汇编语言（因此它有这样的名称）。它还具有使用普通汇编代码语句的能力，这给程序员提供了健壮的平台，帮助他们很容易地从高级语言（比如C或者C++）转移到汇编语言。

HLA的Web站点是http://webster.cs.ucr.edu。Hyde教授使用这个Web站点作为各种汇编器信息的交换场所。这里不仅有很多关于HLA的信息，而且也包括很多其他汇编器包的链接。

### 3.1.2 连接器

如果熟悉高级语言环境，有可能从来都不必直接使用连接器（linker）。很多高级语言（比如C和C++）使用单一命令执行编译和连接两个步骤。

连接目标代码的过程涉及到解析程序代码中声明的所有定义好的函数和内存地址标签。为了达到这一目的，任何外部函数（比如C语言的printf函数）都必须包含在目标代码中（或者提供对外部动态库的引用）。为了自动完成这个工作，连接器必须知道常用目标代码库位于计算机的什么位置，否则必须使用编译器命令行参数来手工指定位置。

但是，大多数汇编器不会自动连接目标代码来生成可执行程序文件。相反，需要第二个手工步骤把汇编语言目标代码和其他库连接在一起，并且生成可以在宿主操作系统上运行的可执行程序文件。这就是连接器的工作。

当手工调用连接器的时候，开发者必须知道完整地解析应用程序使用的所有函数需要哪些库。必须通知连接器函数库在什么位置以及将哪些目标代码文件连接在一起生成最终文件。

每个汇编器包都包含它自己的连接器。总要使用与进行开发使用的汇编器包相匹配的连接器。这有助于确保把函数连接在一起所使用的库文件是相互兼容的，并且输出文件的格式对于目标平台是正确的。

### 3.1.3 调试器

如果是一个完美的程序员，就永远也不需要使用调试器。但是，将来在进行汇编语言程序设计时候，很有可能在什么地方犯错误——要么是10 000行的程序里的打字错误，要么是数学算法函数中的逻辑错误。出现错误时，工具包里有好用的调试器是很方便的。

和汇编器类似，调试器也是为之编写程序的操作系统和硬件平台特定的。调试器必须了解硬件平台的指令码集合并且了解操作系统处理寄存器和内存的方法。

大多数调试器为程序员提供4个基本功能:

- 在一个受控制的环境下运行程序，指定任何必须的运行时参数。
- 在程序内的任何位置停止程序。
- 检查数据元素，比如内存位置和寄存器。
- 在程序运行时改变程序中的元素以便帮助消除缺陷。

调试器在它自己控制下的“沙箱”内运行程序。沙箱允许程序访问内存区域、寄存器和I/O设备，但是这些操作都在调试器的控制之下。调试器能够控制程序如何访问项目并且能够提供信息来显示程序如何以及何时访问项目。

在程序执行过程中的任何位置，调试器都能够停止程序并且指出执行停止在源代码的什么位置。为了做到这一点，调试器必须了解原始源代码，以及从哪些源代码行生成了什么指令码。调试器需要额外的信息被编译到可执行文件中以便识别这些元素。通常在编译或汇编程序时使用特定的命令行参数完成这一任务。

程序在执行过程中停止时，调试器能够显示与程序相关的任何内存区域或者寄存器值。同样，这也是通过在调试器的沙箱内运行程序做到的，这使调试器在程序执行时能够了解程序的内部情况。通过了解各个源代码语句如何影响内存位置和寄存器的值，程序员经常能够发现程序中的错误发生在哪里。这个特性对程序员的价值是无法衡量的。

最后，调试器向程序员提供在程序运行时改变程序中的数据值的途径。这使程序员可以在程序运行时改动程序并且查看这些改动如何影响程序的输出。这是另一个价值无法衡量的特性，它节省了必须改动源代码中的值、重新编译源代码和重新运行可执行文件的时间。

### 3.1.4 编译器

如果你计划使用汇编语言完成所有程序设计工作，那么高级语言编译器就不是必需的。但是，作为专业程序员，可能意识到只使用汇编语言创建成熟的应用程序虽然是可能的，但这将是艰巨而繁重的工作。

相反，大多数专业程序员试图使用高级语言（比如C或者C++）完成尽可能多的应用程序编写工作，然后使用汇编语言程序设计的方法集中精力优化出现麻烦的地方。为了做到这些，你必须有适当的高级语言编译器。

编译器的工作是把高级语言转换为处理器能够执行的指令码。但是，大多数编译器产生一个中间步骤。编译器没有直接把源代码转换为指令码，而是把源代码转换为汇编语言代码。然后使用汇编器把汇编语言代码转换为指令码。许多编译器包含汇编器的处理过程，虽然不是所有编译器都如此。

把C或者C++源代码转换为汇编语言之后，GNU编译器使用GNU汇编器生成连接器使用的指令码。可以在这两个步骤之间停下来，检查从C或者C++源代码生成的汇编语言代码。如果认为可以优化某些部分，那么可以修改生成的汇编语言代码，然后把代码汇编为新的指令码。

### 3.1.5 目标代码反汇编器

试图优化高级语言时，了解代码如何在处理器上运行通常是有帮助的。为了达到这个目的，

需要一种工具来查看编译器从高级语言源代码生成的指令码。在对生成的汇编语言代码进行汇编之前，GNU编译器允许查看这些汇编语言代码，但是如果目标文件已经创建好之后该怎么做呢？

反汇编器程序（disassembler program）处理完整的可执行程序或者目标代码文件，并且显示将运行在处理器上的指令码。一些反汇编器甚至更进了一步，可以把指令码转换为容易阅读的汇编语言语法。

查看编译器生成的指令码之后，可以确定编译器生成的指令码是否充分地优化了。如果不是，能够创建自己的指令码函数来替换编译器生成的函数，以便提高应用程序的性能。

### 3.1.6 简档器

如果在C或者C++环境下进行程序设计，就会经常需要确定程序中花费执行时间最多的函数是哪些。通过查找处理密集型的函数，可以缩小值得花时间去优化的函数的范围。花几天的时间优化一个只占用5%的程序处理时间的功能是在浪费时间。

为了确定处理每个函数占用多长时间，工具包里需要一个简档器（profiler）。简档器能够跟踪每个函数在程序执行过程中被使用时花费了多长处理器时间。

为了优化程序，找到占用最多时间的函数是哪些之后，可以使用反汇编器查看生成的指令码是什么。分析使用的算法确保它们经过优化之后，可以使用编译器没有用到的高级处理器指令手工地生成指令码以便优化函数。

## 3.2 GNU汇编器

GNU汇编器（称为gas）是UNIX环境下最流行的汇编器。它具有为几种不同硬件平台汇编指令码的能力，这些平台包括：

- VAX
- AMD 29K
- Hitachi H8/300
- Intel 80960
- M680x0
- SPARC
- Intel 80x86
- Z8000
- MIPS

本书中所有汇编语言范例都是针对gas编写的。很多UNIX系统在安装的操作系统程序中都包含gas。大多数Linux版本的开发工具实现的默认设置都包含它。

本节讲解如何下载和安装gas，以及如何使用它创建和汇编汇编语言程序。

### 3.2.1 安装汇编器

和大多数其他开发软件包不同，GNU汇编器不在单独的包中发布。它和GNU binutils包中的其他开发软件捆绑在一起。

读者或许需要，或许不需要binutils包中包含的所有子包，但是把它们都安装在系统上不是个坏主意。下表显示当前binutils包（版本2.15）安装的所有程序：

| 包 | 描　述 |
|---|---|
| addr2line | 把地址转换为文件名和行号 |
| ar | 创建、修改和展开文件存档 |
| as | 把汇编语言代码汇编为目标代码 |
| c++filt | 还原C++符号的过滤器 |
| gprof | 显示程序简档信息的程序 |
| ld | 把目标代码文件转换为可执行文件的连接器 |
| nlmconv | 把目标代码转换为Netware Loadable Module格式 |
| nm | 列出目标文件中的符号 |
| objcopy | 复制和翻译目标文件 |
| objdump | 显示来自目标文件的信息 |
| ranlib | 生成存档文件内容的索引 |
| readelf | 按照ELF格式显示来自目标文件的信息 |
| size | 列出目标文件或者存档文件的段长度 |
| strings | 显示目标文件中的可打印字符串 |
| strip | 丢弃符号 |
| windres | 编译Microsoft Windows资源文件 |

大多数支持软件开发的Linux版本已经包含binutils包（特别是当版本包含GNU C编译器时）。可以使用特定Linux版本的包管理器检查binutils包。我的Mandrake Linux系统使用RedHat包管理（RedHat Package Management，RPM）安装包，在这个系统上，使用下面的命令检查binutils。

```
$ rpm -qa | grep binutils
libbinutils2-2.10.1.0.2-4mdk
binutils-2.10.1.0.2-4mdk
$
```

rpm查询命令的输出显示为binutils安装了两个RPM包。第一个包是libbinutils2，它安装binutils包必须的低层库。第二个包是binutils，它安装实际的包。这个系统上可用的包的版本是2.10。

如果使用基于Debian包安装器的Linux版本，可以使用dpkg命令查询已经安装的包：

```
$ dpkg -l | grep binutil
ii  binutils       2.14.90.0.7-3  The GNU assembler, linker and binary utilities
ii  binutils-doc   2.14.90.0.7-3  Documentation for the GNU assembler, linker
$
```

输出显示这个Linux系统上安装了版本2.14的binutils包。

如果在你的Linux系统上已经安装并且正在使用binutils包，通常建议不要改动它。binutils包包含很多用于编译操作系统组件的低层库文件。如果这些库文件被改动或者删除了，那么对你的系统会有不利的影响，非常不利的影响。

如果你的系统没有包含binutils包，可以从binutils的Web站点下载它，网址是http://sources.redhat.com/binutils。这个Web页面包含到binutils下载页面（ftp://ftp.gnu.org/gnu/binutils/）的链接。从那里可以下载binutils当前版本的源代码。编写本书时binutils的版本是2.15，下载文件名

为binutils-2.15.tar.gz。

下载了安装包之后，在一个工作目录里解压缩它，使用下面的命令：

```
tar -zxvf binutils-2.15.tar.gz
```

这个命令在当前目录下面创建名为binutils-2.15的工作目录。为了编译binutils包，转换到这个工作目录并且使用下面的命令：

```
./configure
make
```

configure命令检查宿主系统以便确保编译包所需的所有包和工具在系统上是可用的。编译好软件包之后，可以使用make install命令把软件安装在公共区域便于其他人使用。

### 3.2.2 使用汇编器

GNU汇编器是面向命令行的程序。应该使用正确的命令行参数从命令提示行运行它。这个汇编器的一个奇怪的地方是虽然它称为gas，但是命令行可执行程序却称为as。

根据操作系统使用的是什么硬件平台，as的命令行参数是不同的。对于所有硬件平台都通用的命令行参数如下：

```
as [-a[cdhlns][=file]] [-D] [--defsym sym=val]
   [-f] [--gstabs] [--gstabs+] [--gdwarf2] [--help]
   [-I dir] [-J] [-K] [-L]
   [--listing-lhs-width=NUM] [--listing-lhs-width2=NUM]
   [--listing-rhs-width=NUM] [--listing-cont-lines=NUM]
   [--keep-locals] [-o objfile] [-R] [--statistics] [-v]
   [-version] [--version] [-W] [--warn] [--fatal-warnings]
   [-w] [-x] [-Z] [--target-help] [target-options]
   [--|files ...]
```

下表解释这些命令行参数：

| 参 数 | 描 述 |
|---|---|
| -a | 指定输出中包含哪些清单 |
| -D | 包含它用于向下兼容，但是被忽略了 |
| --defsym | 在汇编源代码之前定义符号和值 |
| -f | 快速汇编，跳过注释和空白 |
| --gstabs | 包含每行源代码的调试信息 |
| --gstabs+ | 包含专门的gdb调试信息 |
| -I | 指定搜索包含文件的目录 |
| -J | 不警告带符号溢出 |
| -K | 包含它用于向下兼容，但是被忽略了 |
| -L | 在符号表中保存本地符号 |
| --listing-lhs-width | 设置输出数据列的最大宽度 |
| --listing-lhs-width2 | 设置连续行的输出数据列的最大宽度 |
| --listing-rhs-width | 设置输入源代码行的最大宽度 |
| --listing-cont-lines | 设置输入的单一行在清单中输出的最大行数 |
| -o | 指定输出目标文件的名称 |
| -R | 把数据段合并进文本段 |

（续）

| 参　数 | 描　述 |
| --- | --- |
| --statistics | 显示汇编使用的最大空间和总时间 |
| -v | 显示as的版本号 |
| -W | 不显示警告消息 |
| -- | 对于源文件使用标准输入 |

下面是把汇编语言程序test.s转换为目标文件test.o的一个例子：

```
as -o test.o test.s
```

这个命令创建目标文件test.o，其中包含汇编语言程序的指令码。如果程序中出现了任何错误，汇编器会通知并且指出问题出在源代码中的什么位置：

```
$ as -o test.o test.s
test.s: Assembler messages:
test.s:16: Error: no such instruction: 'mpvl $4,%eax'
$
```

前面的错误消息特别地指出第16行发生错误并且显示这行的文本。哎呀，看来第16行有打字错误。

### 3.2.3　关于操作码语法

GNU汇编器一个比较容易引起混淆的部分是它在源代码文件中用来表示汇编语言代码的语法。gas的初始开发者选择为汇编器实现AT&T操作码语法。

AT&T操作码语法起源于AT&T贝尔实验室（AT&T Bell Lab），UNIX操作系统是在这里创建的。它是在当时用于实现UNIX操作系统的较流行的处理器芯片的操作码语法之上形成的。虽然很多处理器生产商使用这种格式，但是很不幸，Intel选择使用不同的操作码语法。

出于这个原因，使用gas创建用于Intel平台的汇编语言程序可能是棘手的。大多数关于Intel汇编语言程序设计的文档使用Intel语法，而大多数为旧式UNIX系统编写的文档使用AT&T语法。这给gas程序员带来了混乱和额外的工作。

大多数区别出现在特定的指令格式中，这些区别将在本书讨论指令时讲解。Intel和AT&T语法的主要区别如下：

- AT&T使用$表示立即操作数，而Intel的立即操作数是不需界定的。因此，使用AT&T语法引用十进制值4时，使用$4，使用Intel语法时只需使用4。
- AT&T在寄存器名称前面加上前缀%，而Intel不这样做。因此，使用AT&T语法引用EAX寄存器写为%eax。
- AT&T语法处理源和目标操作数时使用相反的顺序。把十进制值4传送给EAX寄存器，AT&T的语法是movl $4，% eax，而Intel语法是mov eax，4。
- AT&T语法在助记符后面使用一个单独的字符来引用操作中使用的数据长度，而Intel语法中数据长度被声明为单独的操作数。AT&T的指令movl $test，%eax等同于Intel语法的mov eax，dword ptr test。

- 长调用和跳转使用不同语法定义段和偏移值。AT&T语法使用ljmp $section, $offset，而Intel语法使用jmp section:offset。

虽然这些区别使两种格式之间的切换变得困难，但是如果坚持使用其中一种就没有问题了。如果使用AT&T语法学习汇编语言编程，那么在大多数UNIX系统上、在大多数硬件平台上创建汇编语言程序是会感到舒服的。如果计划在UNIX和Microsoft Windows系统之间进行跨平台的工作，则要考虑使用Intel语法编写应用程序。

GNU汇编器提供了使用Intel语法替换AT&T语法的方法，但是在编写本书时，这一功能有些不太好用而且几乎没有文档说明。汇编语言程序中的.intel_syntax命令通知as使用Intel语法汇编指令码助记符，而不使用AT&T语法。不幸的是，这一方法仍然存在很多局限性。例如，尽管源和目标程序被切换为Intel语法，还是必须为寄存器名称加上百分号前缀（就像AT&T语法）。希望未来的as版本能够支持完整的Intel语法汇编代码。

本书中所有汇编语言程序都使用AT&T语法。

## 3.3 GNU连接器

GNU连接器ld用于把目标代码文件连接为可执行程序文件或者库文件。ld程序也是GNU binutils包的一部分，所以如果已经安装了GNU汇编器，那么很可能也安装了连接器。

ld的命令格式如下：

```
ld [-o output] objfile...
   [-Aarchitecture] [-b input-format] [-Bstatic]
   [-Bdynamic] [-Bsymbolic] [-c commandfile] [--cref]
   [-d|-dc|-dp]
   [-defsym symbol=expression] [--demangle]
   [--no-demangle] [-e entry] [-embedded-relocs] [-E]
   [-export-dynamic] [-f name] [--auxiliary name]
   [-F name] [--filter name] [-format input-format]
   [-g] [-G size] [-h name] [-soname name] [--help]
   [-i] [-lar] [-Lsearchdir] [-M] [-Map mapfile]
   [-m emulation] [-n|-N] [-noinhibit-exec]
   [-no-keep-memory] [-no-warn-mismatch] [-Olevel]
   [-oformat output-format] [-R filename] [-relax]
   [-r|-Ur] [-rpath directory] [-rpath-link directory]
   [-S] [-s] [-shared] [-sort-common]
   [-split-by-reloc count] [-split-by-file]
   [-T commandfile]
   [--section-start sectionname=sectionorg]
   [-Ttext textorg] [-Tdata dataorg] [-Tbss bssorg]
   [-t] [-u sym] [-V] [-v] [--verbose] [--version]
   [-warn-common] [-warn-constructors]
   [-warn-multiple-gp] [-warn-once]
   [-warn-section-align] [--whole-archive]
   [--no-whole-archive] [--wrap symbol] [-X] [-x]
```

虽然看上去有很多命令行参数，实际上应该用不着一次使用非常多的参数。这表明GNU连接器是功能极多的程序，并且具有很多不同的能力。下表介绍用于Intel平台的命令行参数。

在最简单的情况下，要从汇编器生成的目标文件创建可执行文件，可以使用下面的命令：

```
Id -o mytest mytest.o
```

这个命令从目标代码文件test.o创建可执行文件test。创建出的可执行文件具有适当的许可，

所以可以在UNIX的控制台中从命令行运行它。这里是该过程的一个例子:

```
$ld -o test test.o
$ ls -al test
-rwxr-xr-x    1 rich     rich          787 Jul  6 11:53 test
$ ./test
Hello world!
$
```

连接器自动使用UNIX 755模式访问来创建可执行文件，这允许使用系统的任何人运行它，但是只有文件的所有者才能修改。

| 参　数 | 描　述 |
|---|---|
| -b | 指定目标代码输入文件的格式 |
| -Bstatic | 只使用静态库 |
| -Bdynamic | 只使用动态库 |
| -Bsymbolic | 把引用捆绑到共享库中的全局符号 |
| -c | 从指定的命令文件读取命令 |
| --cref | 创建跨引用表 |
| -d | 设置空格给通用符号，即使指定了可重定位输出 |
| -defsym | 在输出文件中创建指定的全局符号 |
| --demangle | 在错误消息中还原符号名称 |
| -e | 使用指定的符号作为程序的初始执行点 |
| -E | 对于ELF格式文件，把所有符号添加到动态符号表 |
| -f | 对于ELF格式共享对象，设置DT_AUXILIARY名称 |
| -F | 对于ELF格式共享对象，DT_FILTER名称 |
| -format | 指定目标代码输入文件的格式（和-b相同） |
| -g | 被忽略。用于提供和其他工具的兼容性 |
| -h | 对于ELF格式共享对象，设置DT_SONAME名称 |
| -i | 执行增量连接 |
| -l | 把指定的存档文件添加到要连接的文件清单 |
| -L | 把指定的路径添加到搜索库的目录清单 |
| -M | 显示连接映射，用于诊断目的 |
| -Map | 创建指定的文件来包含连接映射 |
| -m | 模拟指定的连接器 |
| -N | 指定读取/写入文本和数据段 |
| -n | 设置文本段为只读 |
| -noinhibit-exec | 生成输出文件，即使出现非致命连接错误 |
| -no-keep-memory | 为内存使用优化连接 |
| -no-warn-mismatch | 允许连接不匹配的目标文件 |
| -O | 生成优化了的输出文件 |
| -o | 指定输出文件的名称 |
| -oformat | 指定输出文件的二进制格式 |
| -R | 从指定的文件读取符号名称和地址 |
| -r | 生成可重定位的输出（称为部分连接） |
| -rpath | 把指定的目录添加到运行时库搜索路径 |
| -rpath-link | 指定搜索运行时共享库的目录 |
| -S | 忽略来自输出文件的调试器符号信息 |

（续）

| 参 数 | 描 述 |
| --- | --- |
| -s | 忽略来自输出文件的所有符号信息 |
| -shared | 创建共享库 |
| -sort-common | 在输出文件中不按照长度对符号进行排序 |
| -split-by-reloc | 按照指定的长度在输出文件中创建额外的段 |
| -split-by-file | 为每个目标文件在输出文件中创建额外的段 |
| --section-start | 在输出文件中指定的地址定位指定的段 |
| -T | 指定命令文件（和-c相同） |
| -Ttext | 使用指定的地址作为文本段的起始点 |
| -Tdata | 使用指定的地址作为数据段的起始点 |
| -Tbss | 使用指定的地址作为bss段的起始点 |
| -t | 在处理输入文件时显示它们的名称 |
| -u | 强制指定符号在输出文件中作为未定义符号 |
| -warn-common | 当一个通用符号和另一个通用符号结合时发出警告 |
| -warn-constructors | 如果没有使用任何全局构造器，则发出警告 |
| -warn-once | 对于每个未定义符号只发出一次警告 |
| -warn-section-align | 如果为了对齐而改动了输出段地址，则发出警告 |
| --whole-archive | 对于指定的存档文件，在存档中包含所有文件 |
| -X | 删除所有本地临时符号 |
| -x | 删除所有本地符号 |

## 3.4 GNU编译器

GNU编译器集合（GNU Compiler Collection，gcc）是UNIX系统上最流行的开发系统。它不仅是Linux和大多数开放源码的基于BSD的系统（比如FreeBSD和NetBSD）的默认编译器，在很多商业UNIX版本上也很流行。

gcc能够编译很多不同的高级语言。编写本书时，gcc能够编译的高级语言有下面这些：

- C
- C++
- Objective-C
- Fortran
- Java
- Ada

gcc不仅提供编译C和C++应用程序的功能，它还提供了在系统上运行C和C++应用程序所需的库。下面几节介绍如何在系统上安装gcc以及如何使用它来编译高级语言程序。

### 3.4.1 下载和安装gcc

很多UNIX系统已经包含了C开发环境，这是默认安装的。编译C和C++程序需要gcc包。在使用RPM的系统中，可以使用下列命令检查gcc：

```
$ rpm -qa | grep gcc
gcc-cpp-2.96-0.48mdk
gcc-2.96-0.48mdk
gcc-c++-2.96-0.48mdk
$
```

以上信息显示已经安装了gcc C和C++编译器，版本是2.96。如果系统没有安装gcc包，应该首先查看Linux安装光盘。如果Linux版本捆绑了某个版本的gcc，最容易的做法是从这里安装它。和binutils包一样，gcc包括很多库，这些库必须和运行在系统上的程序兼容，否则会发生问题。

如果使用的是不带有gcc包的UNIX系统，可以从gcc的Web站点下载gcc二进制文件（记住，如果没有编译器，就不能编译源代码）。gcc的主页是http://gcc.gnu.org。

在编写本书时，最新的gcc版本是3.4.0。如果所用平台不能使用当前gcc包的二进制版本，可以下载旧一些的版本开始工作，然后下载最新版本的完整源代码。

### 3.4.2　使用gcc

可以使用几种不同的命令行格式调用GNU编译器，使用什么格式取决于要编译的源代码和操作系统的底层硬件。一般的命令行格式如下：

```
gcc [-c|-S|-E] [-std=standard]
    [-g] [-pg] [-Olevel]
    [-Wwarn...] [-pedantic]
    [-Idir...] [-Ldir...]
    [-Dmacro[=defn]...] [-Umacro]
    [-foption...] [-mmachine-option...]
    [-o outfile] infile...
```

下表介绍一般的参数：

| 参　数 | 描　述 |
|---|---|
| -c | 编译或者汇编代码，但是不进行连接 |
| -S | 编译后停止，但是不进行汇编 |
| -E | 预处理后停止，但是不进行编译 |
| -o | 指定要使用的输出文件名 |
| -v | 显示每个编译阶段使用的命令 |
| -std | 指定使用的语言标准 |
| -g | 生成调试信息 |
| -pg | 生成gprof制作简档要使用的额外代码 |
| -O | 优化可执行代码 |
| -W | 设置编译器警告消息级别 |
| -pedantic | 按照C标准发布强制性诊断清单 |
| -I | 指定包含文件的目录 |
| -L | 指定库文件的目录 |
| -D | 预定义源代码中使用的宏 |
| -U | 取消任何定义了的宏 |
| -f | 指定用于控制编译器行为的选项 |
| -m | 指定与硬件相关的选项 |

同样，可以使用很多命令行参数控制gcc的行为。在大多数情况下，只需要使用少数几个参数。如果计划使用调试器监视程序，就必须使用-g参数。对于Linux系统，-gstabs参数在程序中为GNU调试器提供额外的调试信息（在后面3.5.2节中讨论）。

为了测试编译器，可以创建一个简单的C语言程序：

```
#include <stdio.h>
int main()
{
   printf("Hello, world!\n");
   exit(0);
}
```

可以使用下面的命令编译和运行这个简单的C程序：

```
$ gcc -o ctest ctest.c
$ ls -al ctest
-rwxr-xr-x    1 rich     rich        13769 Jul  6 12:02 ctest*
$ ./ctest
Hello, world!
$
```

正如我们期望的，gcc编译器创建了名为ctest的可执行程序文件，并且给赋予它适当的许可以便能够执行（注意这个格式不创建中间目标文件）。程序运行时，它在控制台生成我们期望的输出。

gcc中一个非常有用的命令行参数是-S参数。这个参数让编译器创建中间汇编语言文件，然后汇编器进行汇编。下面是使用-S参数的一个输出例子：

```
$ gcc -S ctest.c
$ cat ctest.s
        .file   "ctest.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "Hello, world!\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        subl    $12, %esp
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        subl    $12, %esp
        pushl   $0
        call    exit
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
$
```

ctest.s文件显示编译器如何创建实现C源代码程序的汇编语言指令。当试图优化C应用程序时，这对于确定编译器如何在指令码中实现各种C语言函数是很有用的。读者也许注意到生成的汇编语言程序使用了两个C函数——printf和lexit。在第4章“汇编语言程序范例”中，将了解汇编语言程序可以怎样简单地使用已经安装在系统中的C库函数。

## 3.5 GNU调试器程序

很多专业程序员使用GNU调试器程序（gdb）调试C和C++应用程序以及查找程序中的错误。读者可能不知道也可以使用gdb调试汇编语言程序。本节讲解gdb包，包括如何下载、安装和使用它的基础功能。本书使用它作为汇编语言应用程序的调试器工具。

### 3.5.1 下载和安装gdb

gdb常常是Linux和BSD开发系统的标准组件。可以使用适当的包管理器确定系统上是否已经安装了它:

```
$ rpm -qa | grep gdb
libgdbm1-1.8.0-14mdk
libgdbm1-devel-1.8.0-14mdk
gdb-5.0-11mdk
$
```

这个Mandrake Linux系统安装了5.0版本的gdb包，还有gdb使用的两个库包。

如果使用的系统没有安装gdb，那么可以从它的Web站点下载它: www.gnu.org/software/gdb/gdb.html。在编写本书时，最新的gdb版本是6.1.1，可以从ftp://sources.redhat.com/pub/gdb/releases下载，文件名是gdb-6.1.tar.gz。

下载发布文件之后，可以使用下面的命令把它解压到工作目录:

```
tar -zxvf gdb-6.1.tar.gz
```

这个命令创建目录gdb-6.1，里面包括所有源代码文件。为了编译这个包，进入工作目录并且使用下面的命令:

```
./configure
make
```

这个命令把源代码文件编译为必需的库文件和gdb可执行文件。可以使用make install命令安装这些文件。

### 3.5.2 使用gdb

GNU调试器命令行程序称为gdb。可以使用几个不同的参数运行它以便修改它的行为。gdb的命令行格式如下:

```
gdb [-nx] [-q] [-batch] [-cd=dir] [-f] [-b bps] [-tty=dev]
    [-s symfile] [-e prog] [-se prog] [-c core] [-x cmds] [-d dir]
    [prog[core|procID]]
```

下表介绍命令行参数:

| 参 数 | 描 述 |
| --- | --- |
| -b | 设置远程调试时串行接口的线路速度 |
| -batch | 以批处理模式运行 |
| -c | 指定要分析的核心转储文件 |
| -cd | 指定工作目录 |
| -d | 指定搜索源文件的目录 |
| -e | 指定要执行的文件 |
| -f | 调试时以标准格式输出文件名和行号 |
| -nx | 不执行来自.gdbinit文件的命令 |
| -q | 安静模式——不输出介绍 |
| -s | 指定符号的文件名 |
| -se | 指定符号和要执行的文件名 |
| -tty | 设置标准输入和输出设备 |
| -x | 从指定的文件执行gdb命令 |

要想使用调试器，就必须使用-gstabs选项编译或者汇编可执行文件，这样使可执行文件内包含必需的信息，以便调试器知道指令码与源代码文件中什么位置是相关联的。gdb启动之后，它使用一个命令行界面接收调试命令：

```
$ gcc -gstabs -o ctest ctest.c
$ gdb ctest
GNU gdb 5.0mdk-11mdk Linux-Mandrake 8.0
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-mandrake-linux"...
(gdb)
```

在gdb命令提示下，可以输入调试命令。可以使用的命令的清单很长。下表列出了一些比较有用的命令。

| 命 令 | 描 述 |
| --- | --- |
| break | 在源代码中设置断点以便停止执行 |
| watch | 设置监视点，当变量到达特定值时停止执行 |
| info | 观察系统元素，比如寄存器、堆栈和内存 |
| x | 检查内存位置 |
| print | 显示变量值 |
| run | 在调试器内开始程序的执行 |
| list | 列出指定的函数或者行 |
| next | 执行程序中的下一条指令 |
| step | 执行程序中的下一条指令 |
| cont | 从停止的位置继续执行程序 |
| until | 运行程序，直到到达指定的源代码行（或者更大的） |

下面是gdb会话的一个简短的例子：

```
(gdb) list
1       #include <stdio.h>
2
3       int main()
4       {
5               printf("Hello, world!\n");
6               exit(0);
7       }
(gdb) break main
Breakpoint 1 at 0x8048496: file ctest.c, line 5.
(gdb) run
Starting program: /home/rich/palp/ctest

Breakpoint 1, main () at ctest.c:5
5               printf("Hello, world!\n");
(gdb) next
Hello, world!
6               exit(0);
(gdb) next

Program exited normally.
(gdb)quit
$
```

首先，使用list命令显示源代码行号。接下来，使用break命令在main标记处设置断点，然后使用run命令启动程序。因为断点设置在main处，所以程序在main之后的第一个源代码语句之前立即停止。使用next命令执行下一行源代码，即执行printf语句。使用另一个next命令执行exit语句，它终止应用程序。虽然应用程序被终止了，但是仍然在调试器中，并且可以选择再次运行程序。

## 3.6 KDE调试器

GNU调试器是功能非常多的工具，但是它的用户界面还有待相当大的提高。通常，试图使用gdb调试大型应用程序是困难的。为了解决这个问题，几个不同的gdb图形化前端程序被创建出来。其中最流行的一个是KDE调试器（kdbg），创建者是Johannes Sixt。

kdbg包使用K桌面环境（K Desktop Environment，KDE）平台，KDE是主要使用在开放源代码的UNIX系统（比如Linux）上的X-windows图形环境，在其他UNIX平台上也是可用的。它是使用Qt图形库开发的，所以系统上还必须安装Qt运行时库。

### 3.6.1 下载和安装kdbg

很多Linux版本包含kdbg包作为附加包，默认情况下是不安装的。可以检查Linux系统上的发布包管理器，查看这个包是否已经安装了，或者是否可以从Linux发布光盘上安装它。在我的Mandrake系统上，它包含在补充程序光盘上：

```
$ ls kdbg*
kdbg-1.2.0-0.6mdk.i586.rpm
$
```

如果Linux系统上还没有安装这个包，可以从kdbg的Web站点http://members.nextra.at/johsixt/kdbg.html下载kdbg包的源代码。在编写本书时，最新的kdbg稳定版本是1.2.10。最新的测试版

本是1.9.5。

kdbg源代码的安装要求KDE开发头文件是可用的。这些文件通常包含在Linux发布版附带的KDE开发包中，并且也许已经安装好了。

### 3.6.2 使用kdbg

安装好kdbg之后，可以这样使用它：从KDE桌面打开命令提示窗口，然后输入kdbg命令。启动了kdbg之后，必须使用File菜单项或者工具栏图标选择要调试的可执行文件，还必须选择原始源代码文件。

加载可执行文件和源代码文件之后，就可以开始调试会话了。因为kdbg是gdb的图形界面，所以可以使用相同的命令，但是形式是图形方式。可以通过高亮显示适当的源代码行并且点击工具栏上的stop标记图标在应用程序之内设置断点（见图3-1）。




图 3-1

如果希望在程序执行的过程中监视内存或者寄存器值，可以通过点击View菜单项，然后选择希望查看的那些窗口来选择它们。还可以在程序执行时打开输出窗口查看程序的输出。图3-2是Registers窗口的一个例子。

加载程序并且设置期望查看的窗口之后，可以通过点击run图标按钮启动程序的执行。就像在gdb中一样，程序会执行到第一个断点的位置。到达断点时，可以使用step图标按钮继续执行程序，直到程序结束。

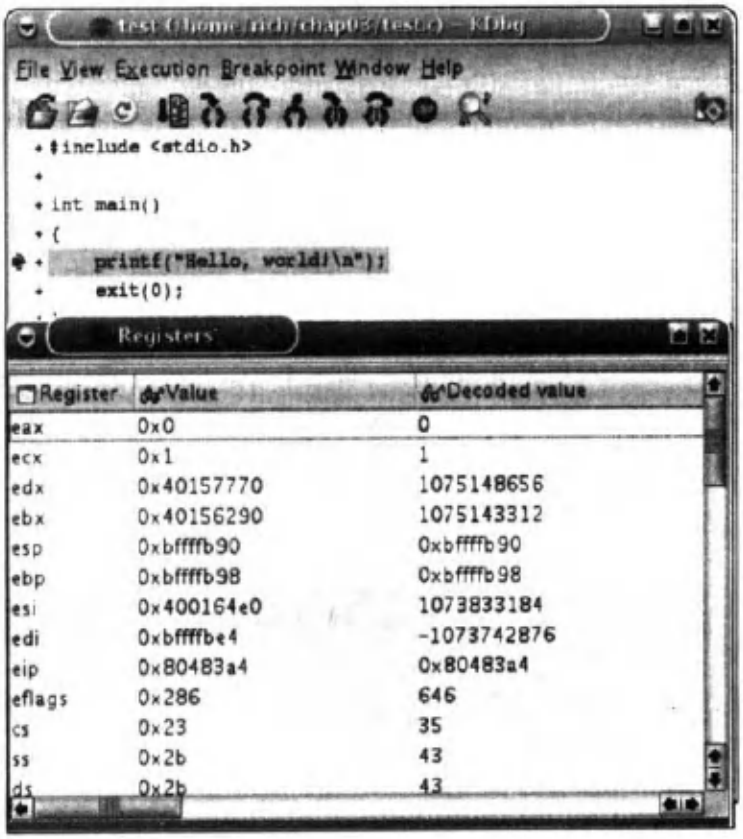



图 3-2

## 3.7 GNU objdump程序

GNU objdump程序是binutils包中另一个对程序员非常有用的工具。程序员经常必须查看目标代码文件中由编译器生成的指令码。objdump程序不仅能够显示汇编语言代码，而且能够显示生成的原始指令码。

本节讲解objdump程序，以及如何使用它查看高级语言程序中包含的底层指令码。

### 3.7.1 使用objdump

objdump的命令行参数指定这个程序对目标代码文件执行什么操作，以及如何显示它获得的信息。objdump的命令行格式如下：

```
objdump [-a|--archive-headers] [-b bfdname|--target=bfdname]
        [-C|--demangle[=style] ] [-d|--disassemble]
        [-D|--disassemble-all] [-z|--disassemble-zeroes]
        [-EB|-EL|--endian={big | little }] [-f|--file-headers]
        [--file-start-context] [-g|--debugging]
        [-e|--debugging-tags] [-h|--section-headers|--headers]
        [-i|--info] [-j section|--section=section]
        [-l|--line-numbers] [-S|--source]
        [-m machine|--architecture=machine]
        [-M options|--disassembler-options=options]
        [-p|--private-headers] [-r|--reloc]
        [-R|--dynamic-reloc] [-s|--full-contents]
        [-G|--stabs] [-t|--syms] [-T|--dynamic-syms]
        [-x|--all-headers] [-w|--wide]
```

```
[--start-address=address] [--stop-address=address]
[--prefix-addresses] [--[no-]show-raw-insn]
[--adjust-vma=offset] [-V|--version] [-H|--help]
objfile...
```

下表介绍命令行参数。

| 参 数 | 描 述 |
|---|---|
| -a | 如果任何文件是存档文件，则显示存档头信息 |
| -b | 指定目标代码文件的目标代码格式 |
| -C | 将低级符号还原为用户级别的名称 |
| -d | 把目标代码反汇编为指令码 |
| -D | 把所有段反汇编为指令码，包括数据 |
| -EB | 指定大尾数目标文件 |
| -EL | 指定小尾数目标文件 |
| -f | 显示每个文件头的摘要信息 |
| -G | 显示调试段的内容 |
| -h | 显示每个文件段头的摘要信息 |
| -i | 显示所有架构和目标格式的清单 |
| -j | 只显示指定段的信息 |
| -l | 使用源代码行号标记输出 |
| -m | 指定进行反汇编时使用的架构 |
| -p | 显示目标文件格式特有的信息 |
| -r | 显示文件中的重定位条目 |
| -R | 显示文件中的动态重定位条目 |
| -s | 显示指定段的完整内容 |
| -S | 交错显示源代码和反汇编后的代码 |
| -t | 显示文件的符号表条目 |
| -T | 显示文件的动态符号表条目 |
| -x | 显示文件所有可用的头信息 |
| --start-address | 开始显示在指定地址上的数据 |
| --stop-address | 停止显示在指定地址上的数据 |

objdump程序是功能非常多的工具。除了目标代码文件之外，它还可以解码很多不同类型的二进制文件。对于汇编语言程序员来说，-d参数是最有意思的，因为它显示反汇编后的目标代码文件。

### 3.7.2 objdump范例

使用范例C程序，可以通过使用-c选项编译程序，创建用于转储的目标文件：

```
$ gcc -c ctest.c
$ objdump -d ctest.o

ctest.o:     file format elf32-i386

Disassembly of section .text:

00000000 <main>:
   0:   55                      push   %ebp
```

```
   1:   89 e5                    mov    %esp,%ebp
   3:   83 ec 08                 sub    $0x8,%esp
   6:   83 ec 0c                 sub    $0xc,%esp
   9:   68 00 00 00 00           push   $0x0
   e:   e8 fc ff ff ff           call   f <main+0xf>
  13:   83 c4 10                 add    $0x10,%esp
  16:   83 ec 0c                 sub    $0xc,%esp
  19:   6a 00                    push   $0x0
  1b:   e8 fc ff ff ff           call   1c <main+0x1c>
$
```

在读者的系统上创建的反汇编后的目标代码文件可能与本例不同，这取决于使用的特定编译器和编译器版本。这个例子显示编译器创建的汇编语言助记符和相应的指令码。但是读者也许注意到了，程序中引用的内存地址被标记为零。在连接器连接应用程序并且使它准备好在系统上执行之前，这些值还不能确定下来。但是在处理的这个阶段，可以很容易地看到使用了什么指令去完成功能。

## 3.8 GNU简档器程序

GNU简档器（gprof）是binutils包中包含的另一个程序。这个程序用于分析程序的执行和确定应用程序中的“热点”在什么位置。

应用程序的热点是程序运行时需要最多处理时间的函数。通常，它们是最为数学密集型的函数，但是情况不总是如此。I/O密集型的函数也会增加处理时间。

本节介绍GNU简档器，并且提供简单的例子，演示在C程序中如何使用它查看应用程序中不同函数花费了多长时间。

### 3.8.1 使用简档器

和其他所有工具一样，gprof是一个命令行程序，使用多个参数控制它的行为。gprof的命令行格式如下：

```
gprof [ -[abcDhilLsTvwxyz] ] [ -[ACeEfFJnNOpPqQZ][name] ]
      [ -I dirs ] [ -d[num] ] [ -k from/to ]
      [ -m min-count ] [ -t table-length ]
      [ --[no-]annotated-source[=name] ]
      [ --[no-]exec-counts[=name] ]
      [ --[no-]flat-profile[=name] ] [ --[no-]graph[=name] ]
      [ --[no-]time=name] [ --all-lines ] [ --brief ]
      [ --debug[=level] ] [ --function-ordering ]
      [ --file-ordering ] [ --directory-path=dirs ]
      [ --display-unused-functions ] [ --file-format=name ]
      [ --file-info ] [ --help ] [ --line ] [ --min-count=n ]
      [ --no-static ] [ --print-path ] [ --separate-files ]
      [ --static-call-graph ] [ --sum ] [ --table-length=len ]
      [ --traditional ] [ --version ] [ --width=n ]
      [ --ignore-non-functions ] [ --demangle[=STYLE] ]
      [ --no-demangle ] [ image-file ] [ profile-file ... ]
```

这一大堆参数分为3组：

- 输出格式参数

- 分析参数
- 杂项参数

下表介绍输出格式参数，它们允许修改gprof生成的输出。

| 参 数 | 描 述 |
|---|---|
| -A | 显示所有函数的源代码，或者只显示指定函数的 |
| -b | 不显示解释分析字段的详细输出 |
| -C | 显示所有函数的总计数，或者只显示指定函数的 |
| -i | 显示简档数据文件的摘要信息 |
| -I | 指定查找源文件的搜索目录清单 |
| -J | 不显示注解的源代码 |
| -L | 显示源文件名的完整路径名称 |
| -p | 显示所有函数的一般简档，或者只显示指定函数的 |
| -P | 不输出所有函数的一般简档，或者不显示指定函数的 |
| -q | 显示调用图表分析 |
| -Q | 不显示调用图表分析 |
| -y | 在单独的输出文件中生成注解的源代码 |
| -Z | 不显示函数的总计数和被调用的次数 |
| --function-reordering | 按照分析显示建议的函数的重排序 |
| --file-ordering | 按照分析显示建议的目标文件重排序 |
| -T | 按照传统的BSD样式显示输出 |
| -w | 设置输出行的宽度 |
| -x | 在函数之内显示被注解的源代码中的每一行 |
| --demangle | 在显示输出时C++符号被还原 |

下表介绍分析参数，它们修改gprof分析包含在分析文件中的数据的方式。

| 参 数 | 描 述 |
|---|---|
| -a | 不分析静态声明（私有）的函数的信息 |
| -c | 分析程序中永远不会被调用的子函数的信息 |
| -D | 忽略已知不是函数的符号（只在Solaris和HP操作系统上） |
| -k | 不分析匹配开头和结尾的symspec的函数 |
| -l | 按行分析程序，而不是按函数 |
| -m | 只分析被调用超过指定次数的函数 |
| -n | 只分析指定的函数的时间 |
| -N | 不分析指定的函数的时间 |
| -z | 分析所有函数，即使是从不被调用的那些 |

最后，下表介绍杂项参数，这些参数也修改gprof的行为，但是不适合放在输出参数组或者分析参数组之内。

| 参 数 | 描 述 |
|---|---|
| -d | 使gprof处于调试模式中，指定数字化的调试级别 |
| -O | 指定简档数据文件的格式 |
| -s | 使gprof只在简档数据文件中汇总数据 |
| -v | 输出gprof的版本 |

为了对应用程序使用gprof，必须确保使用-pg参数编译希望监视的函数。用这个参数编译源代码，就会为程序中的每个函数插入对mcount子例程的调用。当应用程序运行时，mcount子例程创建一个调用图表简档文件，称为gmon.out，它包含应用程序中每个函数的计时信息。

运行应用程序时要小心，因为每次运行都会覆盖gmon.out文件。如果希望进行多次采样，就必须在gprof的命令行中包含输出文件的名称并且在每次采样时使用不同的文件名。

程序测试完成之后，使用gprof程序查看调用图表简档文件，分析每个函数花费的时间。gprof的输出包含3个报告：

- 一般简档报告，它列出总执行时间和所有函数的调用次数。
- 按照每个函数及其子函数花费的时间进行排序的函数清单。
- 循环清单，显示循环成员和它们的调用次数。

默认设置下，gprof的输出直接发送到控制台的标准输出。如果希望保存输出，就必须把它重定向到文件。

### 3.8.2 简档范例

为了演示gprof程序，必须有一个使用函数执行操作的高级语言程序。我使用C语言创建了下面这个简单的演示程序demo.c，用它演示gprof的基本使用：

```
#include <stdio.h>

void function1()
{
        int i, j;
        for(i=0; i <100000; i++)
                j += i;
}
void function2()
{
        int i, j;
        function1();
        for(i=0; i < 200000; i++)
                j = i;
}

int main()
{
        int i, j;
        for (i = 0; i <100; i++)
                function1();

        for(i = 0; i<50000; i++)
                function2();
        return 0;
}
```

这个程序很简单。主程序有两个循环：一个调用function1() 100次；另一个调用function2() 50 000次。每个函数只执行简单的循环，但是每次调用function2()时，它也调用function1 ()。

下一个步骤使用-pg参数编译程序，以便gprof能够执行。编译之后就可以运行这个程序：

```
$ gcc -o demo demo.c -pg
$ ./demo
$
```

程序结束后，在相同的目录下会创建gmon.out调用图表简档文件。然后可以对演示程序运行gprof程序，并且把输出保存到一个文件：

```
$ ls -al gmon.out
-rw-r--r--    1 rich     rich          426 Jul  7 12:39 gmon.out
$ gprof demo > gprof.txt
$
```

注意，命令行中没有引用gmon.out文件，只有可执行程序的名称。gprof自动使用位于相同目录下的gmon.out文件。这个例子把gprof的输出重定向到名为gprof.txt的文件。产生的文件包含程序的完整gprof报告。下面是在我的系统上一般简档部分的样例：

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 67.17     168.81   168.81    50000  3376.20  5023.11  function2
 32.83     251.32    82.51    50100  1646.91  1646.91  function1
```

这个报告显示总的处理器时间和main调用每个函数的调用时间。就像我们预期的，function2占用了主要的处理时间。

下一个报告是调用图表，它显示各个函数的细分时间，以及函数是如何被调用的：

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00  251.32                 main [1]
              168.81   82.35   50000/50000       function2 [2]
                0.16    0.00     100/50100      function1 [3]
-----------------------------------------------
              168.81   82.35   50000/50000       main [1]
[2]     99.9  168.81   82.35   50000          function2 [2]
               82.35    0.00   50000/50100       function1 [3]
-----------------------------------------------
                0.16    0.00     100/50100      main [1]
               82.35    0.00   50000/50100       function2 [2]
[3]     32.8   82.51    0.00   50100          function1 [3]
-----------------------------------------------
```

调用图表的每个部分显示被分析的函数（即带有索引号码的行中的那一个函数）、调用它的函数和它的子函数。使用这个输出跟踪整个程序之内时间流。

## 3.9 完整的汇编开发系统

既然已经了解了汇编语言开发环境需要的所有部件，现在是把它们组合起来的时候了。使用GNU工具的最好的环境是Linux操作系统。很多免费的Linux版本已经安装了本章中出现的所有GNU工具。本节讲解Linux系统的一些基础问题，还有创建汇编语言开发环境所需的GNU工具。

### 3.9.1 Linux基础

如果刚刚接触Linux环境，在尝试某个Linux版本之前，或许需要一些背景信息。当人们谈论

Linux操作系统时，他们实际谈论的是一整套程序，不是所有这些程序都必然和Linux相关。构建Linux系统的关键在于了解构成系统的组件，以及了解从哪里和如何获得它们。

Linus Torvalds创建了Linux操作系统内核并且引导着它的开发。操作系统内核是与硬件进行交互以及处理操作系统的低层功能（比如文件访问控制、处理内存和硬件接口）的软件。仅仅在计算机上加载Linux内核是非常令人厌烦的工作。为了实际地完成任何工作，需要额外的程序来和设备进行交互。

这里就应该提到GNU项目了。多年以来，GNU项目开发了很多应用程序，帮助系统管理员和程序员使用任何UNIX类型的操作系统，其中最流行的系统之一就是Linux。

当下载Linux版本时，下载的是Linux内核，它和一组工具捆绑在一起，这些工具为希望构建的系统类型执行预期的功能。GNU项目已经实现了大多数标准UNIX功能。根据希望特定的Linux系统完成的工作，读者或许希望、或许不希望安装所有可用的GNU程序。

构建好基本的Linux系统之后，会想要创建特定的开发环境。当决定了希望在系统中包含哪些工具之后，首先要查看Linux版本附带的发布光盘是否包含它们。这是目前为止安装包的最简单的方法，特别是在使用的版本包括自动化的包管理器时（比如Red Hat的rpm或者Debian的dpkg）。

如果没有（或者不能）安装完整的Linux系统，那么次优的方法是使用可引导的光盘版本。在可引导的光盘版本中，完整的Linux系统存储在可引导的光盘上。为了运行Linux，只需把光盘放入计算机并且重新启动。Linux系统会加载到内存中，并且在内存中创建虚拟磁盘。这个硬盘驱动器上的操作系统永远都不会被改动。完成之后，把光盘取出，然后从硬盘重新启动。大多数可引导的Linux光盘版本能够出色地完成自动检测工作站硬件（比如网卡、声卡和各种图形卡）的工作。

我最喜欢的用于开发工作的Linux光盘版本之一是MEPIS Linux。MEPIS Linux基于Debian Linux，但是作为可引导光盘，它也可以很容易地安装到硬盘上，如果读者选择这么做的话。这是在现有的工作站系统上安装Linux的最简单的方式之一。

MEPIS是我最喜欢的系统之一的另外一个原因在于，在编写本书的时候，MEPIS可引导光盘包含完整的开发环境，包括gas、ld、gcc、gdb、gprof，甚至还有kdbg。只需从MEPIS光盘进行引导，就具有了一个自动化的Linux汇编语言开发系统!

下面几节讨论如何下载和使用MEPIS Linux版本。

### 3.9.2 下载和运行MEPIS

MEPIS的Web站点是www.mepis.org。从这个页面，可以购买光盘、购买付费下载预订，或者转到免费下载镜像站点。编写本书时，最新的MEPIS完整发布版本是2003-10，patch 2，最新的测试版本是2004-b05。

编写本书时，还有一个单独的MEPIS版本，称为Simply-MEPIS。这个版本包含编译器、汇编器和连接器软件，但是不包含gdb调试器。gdb调试器必须单独下载和安装。

下载的MEPIS光盘的完整版本是两个独立的.iso格式文件：

- mepis-2003-10.02.cd1.iso包含主要的MEPIS软件。
- mepis-2003-10.02.cd2.iso包含Debian包格式的附加包。

如果打算下载发布文件，应该找一个高速因特网连接，因为文件的大小是694MB。下载了.iso文件之后，必须有装有光盘刻录机的工作站，还有能够从.iso文件刻录光盘的软件。把.iso文件刻录为光盘之后，就做好了启动MEPIS Linux的准备。

第一个.iso文件包含可以从光盘运行的完整操作系统。首先确保工作站允许从光盘进行引导（这是系统BIOS中的一个选项）。从MEPIS Linux光盘进行引导时，会出现引导屏幕，要求输入任何专门的引导参数。大多数情况下，只需按下回车键继续引导过程。如果MEPIS不支持所用显卡，可以输入引导提示参数。专门的详细信息参见MEPIS的Web站点。

引导系统之后，会出现KDE桌面登录屏幕，带有两个预先配置的用户ID——demo和root。可以使用其中任何一个帐户进行登录，但是使用demo帐户是最安全的。demo帐户的密码是demo，root帐户的密码是root。

### 3.9.3 新的开发系统

登录到MEPIS系统之后，会看到KDE桌面环境。它类似于Microsoft Windows桌面环境，带有桌面图标、工具栏和Start按钮（虽然在KDE桌面上不称之为Start）。可以通过点击工具栏上的shell图标打开一个命令提示会话。还可以从Editors菜单项打开编辑器会话，使用几种不同编辑器中的任何一种。

创建程序文件时，要注意把文件保存为纯文本文件。MEPIS包含一些奇特的文字处理编辑器，比如OpenOffice Writer，它可以按照特殊的二进制格式保存文档。GNU汇编器不能读取这些格式。

创建好汇编语言程序的文本文件之后，可以从命令提示会话使用as和ld命令对程序进行汇编和连接。如果打算创建高级语言程序，MEPIS也包含用于C和C++应用程序的gcc编译器。

对于调试，MEPIS包含gdb和kdbg程序。可以从命令提示会话访问gdb程序，可以从Development部分下的菜单访问kdbg程序。

从可引导光盘运行开发环境的唯一缺陷出现在保存工作的时候。默认情况下，系统使用RAM内存作为虚拟磁盘。在默认文件系统下存储的任何程序都只存储在内存中。下次从光盘引导时，文件就丢失了。

为了解决这个问题，应该有某种MEPIS能够访问的介质。不幸的是，编写本书时，MEPIS还不能把数据写入使用NTFS格式（通常在Windows 2000和XP工作站上使用）的硬盘中。如果有使用FAT32格式格式化的硬盘，MEPIS能够写入它。另外，MEPIS可以把文件写入软盘，以及大多数USB闪存驱动器。最后一个办法是，如果工作站在局域网（local area network，LAN）上，那么总是可以通过使用FTP协议，或者通过把Windows共享驱动器装载到系统上来复制文件。我喜欢的方法是使用另一台计算机上的Windows共享。这和在Windows中进行映射一样简单。

## 3.10 小结

每个程序员都需要一个开发环境，以便在其中创建应用程序。不幸的是，汇编语言程序员通常必须创建他们自己的开发环境。要把很多不同的组件结合在一起创建完美的开发环境。

至少需要文本编辑器、汇编器包和连接器包（通常汇编器包和连接器包是捆绑在一起的）。

汇编器用于把汇编语言代码转换为用于运行应用程序的特定处理器的指令码。然后，使用连接器，通过组合任何必需的库以及解析用于内存存储的所有内存引用，把原始指令码转换为可执行程序。

除了汇编器和连接器之外，经常还需要调试器和目标代码反汇编器。调试器使得可以按照步进方式执行程序，监视每条指令如何修改寄存器和内存位置。反汇编器使得可以查看由汇编语言程序或者高级语言程序生成的目标代码文件中的指令码。

如果打算把高级语言和汇编代码一起使用，那么还需要编译器，用它从高级语言源代码构建可执行代码。很多编译器也具有显示从源代码生成的指令码的能力，这使得可以查看源代码指令实际上被转换成了什么。这就是真正用得到汇编语言程序设计的地方。通过检查生成的指令码，有时可以发现存在比编译器所做的更好的实现函数的方式，然后自己完成它。

另一个对于程序员有用的工具是简档器。简档器用于分析应用程序的性能。通过检查哪些函数占用了最多的处理时间，可以确定哪些函数值得优化，以便提高应用程序的性能。

虽然这些工具的版本有很多，但是本书使用GNU项目开发的工具。这些工具都是免费的，可以运行在大多数UNIX系统上。GNU的binutils包包含了大多数内核工具。汇编器as、连接器ld、目标代码反汇编器objdump和简档器gprof，都包含在binutils包中。GNU调试器称为gdb，GNU编译器称为gcc。

既然开发环境已经构建完成了，现在是开始进行一些汇编语言程序设计的时候了。下一章介绍如何使用工具创建汇编语言范例程序。

# 第 4 章　汇编语言程序范例

准备好所有开发工具之后，就可以开始学习汇编语言程序设计了。汇编语言程序使用通用模板和格式（是使用的汇编器特定的），可以把它们用于所有应用程序的开发。

本章引导读者学习GNU汇编器的基本汇编语言程序模板。本章的第一节介绍汇编语言程序中的通用项目，以及如何使用它们定义通用模板。下一节讲解一个范例程序，以及如何汇编和运行它。接下来将学习如何使用GNU调试器调试范例程序。本章的最后一节演示如何把C库函数并入汇编语言程序中。

## 4.1　程序的组成

正如第1章“什么是汇编语言”中介绍的，汇编语言程序由定义好的段构成，每个段都有不同的目的。三个最常用的段如下：

- 数据段
- bss段
- 文本段

所有汇编语言程序中都必须有文本段。这里是在可执行程序内声明指令码的地方。数据和bss段是可选的，但是在程序中经常使用。数据段声明带有初始值的数据元素。这些数据元素用作汇编语言程序中的变量。bss段声明使用零（或者null）值初始化的数据元素。这些数据元素最常用作汇编语言程序中的缓冲区。

下面几节介绍如何在为GNU汇编器编写的汇编语言程序中声明不同的段，GNU汇编器是本书中使用的汇编器。

### 4.1.1　定义段

GNU汇编器使用.section命令语句声明段。.section语句只使用一个参数——它声明的段的类型。图4-1显示汇编语言程序的布局。

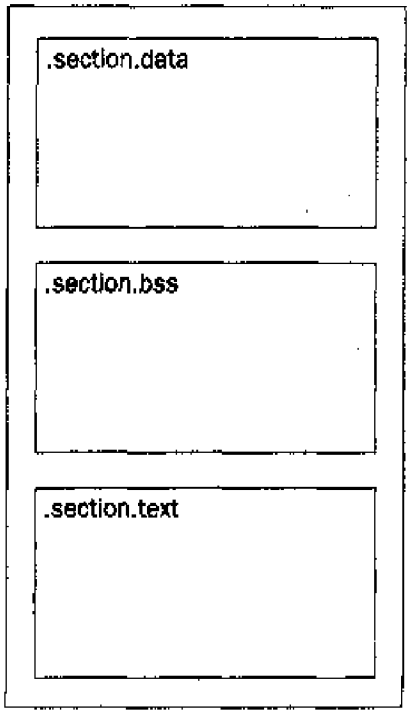



图 4-1

图4-1演示在程序中安排段的一般方式。bss段总是应该安排在文本段之前，但是数据段可以移动到文本段之后，虽然这不是标准。除了完成功能性的要求之外，编写的汇编语言程序还应该容易阅读。将所有数据定义集中在源代码的开头使其他程序员更加容易接手你的工作并且理解它。

### 4.1.2　定义起始点

当汇编语言程序被转换为可执行文件时，连接器必须知道指

令码中的起始点是什么。对于只有单一指令路径的简单程序，找到起始点通常不是问题。但是，对于使用分散在源代码各个位置的若干函数的更加复杂的程序，发现程序从哪里开始可能是个问题。

为了解决这个问题，GNU汇编器声明一个默认标签，或者说标识符，它应该用作应用程序的入口点。_start标签用于表明程序应该从这条指令开始运行。如果连接器找不到这个标签，它会生成错误消息：

```
$ ld -o badtest badtest.o
ld: warning: cannot find entry symbol _start; defaulting to 08048074
$
```

就像连接器的输出显示的，如果连接器找不到_start标签，它就会试图查找程序的起始点，但是对于复杂的程序，不能保证连接器能正确地作出猜测。

也可以使用_start之外的其他标签作为起始点。可以使用连接器的-e参数定义新的起始点的名称。

除了在应用程序中声明起始标签之外，还需要为外部应用程序提供入口点。这是使用.globl命令完成的。

.globl命令声明外部程序可以访问的程序标签。如果编写被外部汇编语言或者C语言程序使用的一组工具，就应该使用.globl命令声明每个函数段标签。

了解了这些信息，就可以为所有汇编语言程序创建基础模板。模板应该像下面这样：

```
.section.data

        <  initialized data here>

.section .bss

        < uninitialized data here>

.section .text
.globl _start
_start:

    <instruction code goes here>
```

有了这样的模板，就准备好了开始编写汇编语言程序。下一节讲解一个范例应用程序，它展示如何从汇编语言程序源代码构建应用程序。

## 4.2 创建简单程序

现在是创建简单的汇编语言应用程序来演示这些组件如何结合在一起使用的时候了。最开始，我们创建一个重点在单一指令码的简单应用程序。CPUID指令码用于收集程序正在其上运行的处理器的信息。可以从处理器获取生产厂商和型号的信息，并且把信息显示给用户。

下面几节介绍CPUID指令以及如何使用它来实现汇编语言程序。

### 4.2.1 CPUID指令

CPUID指令是一条汇编语言指令，不容易从高级语言应用程序执行它。它是请求处理器的

特定信息并且把信息返回到特定寄存器中的低级指令。

CPUID指令使用单一寄存器值作为输入。EAX寄存器用于决定CPUID指令生成什么信息。根据EAX寄存器的值，CPUID指令在EBX、ECX和EDX寄存器中生成关于处理器的不同信息。信息以一系列位值和标志的形式返回，必须解释出它们的正确含义。

下表介绍CPUID指令可用的不同输出选项。

| EAX值 | CPUID输出 |
|---|---|
| 0 | 厂商ID（Vendor ID）字符串和支持的最大CPUID选项值 |
| 1 | 处理器类型、系列、型号和分步信息 |
| 2 | 处理器缓存配置 |
| 3 | 处理器序列号 |
| 4 | 缓存配置（线程数量、核心数量和物理属性） |
| 5 | 监视信息 |
| 80000000h | 扩展的厂商ID字符串和支持的级别 |
| 80000001h | 扩展的处理器类型、系列、型号和分步信息 |
| 80000002h - 80000004h | 扩展的处理器名称字符串 |

本章中创建的范例程序使用零选项从处理器获得简单的厂商ID字符串。当零值被放入EAX寄存器并且执行CPUID指令时，处理器把厂商ID字符串返回到EBX、EDX和ECX寄存器中，如下：

- EBX包含字符串的最低4个字节。
- EDX包含字符串的中间4个字节。
- ECX包含字符串的最高4个字节。

字符串值按照小尾数格式放在寄存器中；因此，字符串的第一部分放在寄存器的低位中。图4-2显示这种情况。

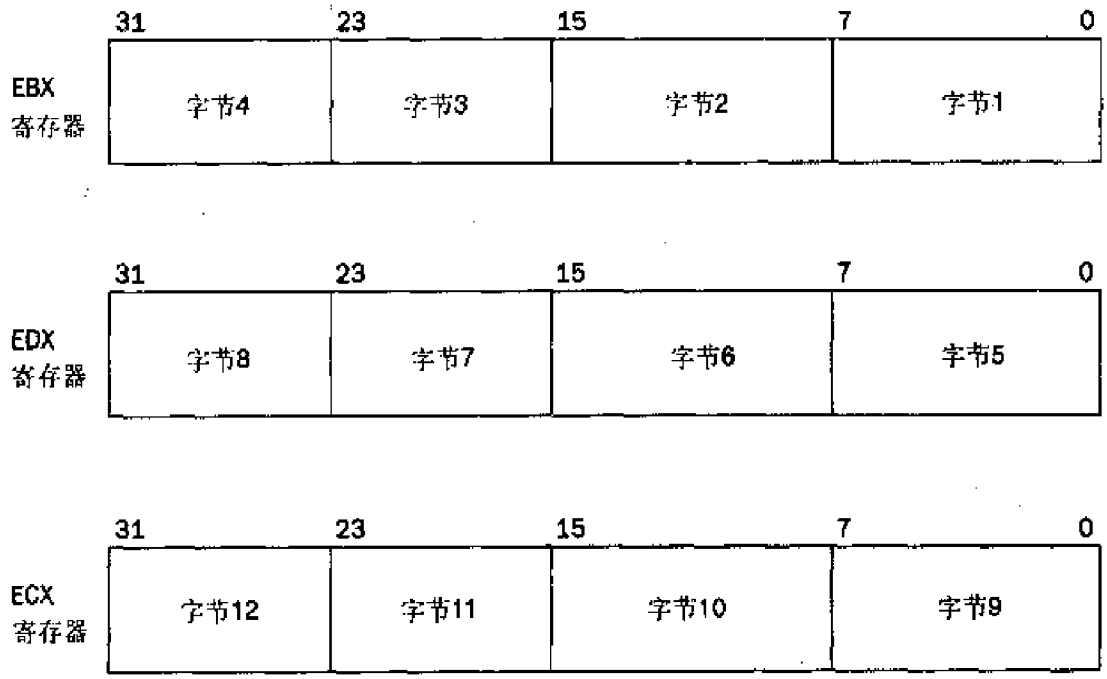



图 4-2

范例程序获得寄存器值并且按照易读的格式把信息显示给用户。下一节将给出这个范例程序。

并不是所有IA-32平台的处理器都按照相同的方式使用CPUID指令。在实际的应用程序中，应该进行一些测试以便确保处理器支持CPUID指令。为了使讲解简单，本章中介绍的范例程序没有进行这些测试。有可能读者使用的处理器不支持CPUID指令，虽然大多数现代的处理器确实支持它（包括Intel的奔腾处理器、Cyrix处理器和AMD处理器）。

### 4.2.2 范例程序

了解了CPUID指令如何工作之后，现在是开始编写简单的程序来使用这些信息的时候了。这个程序是一个简单的应用程序，它查看CPUID指令生成的厂商ID字符串。下面就是范例程序cpuid.s:

```
#cpuid.s Sample program to extract the processor Vendor ID
.section .data
output:
   .ascii "The processor Vendor ID is 'xxxxxxxxxxxx'\n"
.section .text
.globl _start
_start:
   movl $0, %eax
   cpuid
movl $output, %edi
movl %ebx, 28(%edi)
movl %edx, 32(%edi)
movl %ecx, 36(%edi)
movl $4, %eax
movl $1, %ebx
movl $output, %ecx
movl $42, %edx
int $0x80
movl $1, %eax
movl $0, %ebx
int $0x80
```

这个程序使用了许多不同的汇编语言指令。现在，不必过于担心这些语句是什么；以后的章节中将详细地讲解它们。目前请把注意力集中在如何在程序中安排指令、指令如何操作的流程以及源代码文件如何转换为可执行程序文件上。这样就不会给自己造成混乱，下面是对源代码如何工作的简要讲解。

首先，在数据段中声明了一个字符串:

```
output:
   .ascii "The processor Vendor ID is 'xxxxxxxxxxxx'\n"
```

.ascii声明使用ASCII字符声明一个文本字符串。字符串元素被预定义并且放在内存中，其起始内存位置由标签output指示。后面的x在保留给数据变量的内存区域中作为占位符。从处理器获得厂商ID字符串时，会把它放在位于这些内存位置的数据中。

读者应该认识程序模板中接下来的段落。它声明程序的指令码段和一般的起始标签:

```
.section .text
.globl _start
_start:
```

程序做的第一件事情是使EAX寄存器加载零值，然后运行CPUID指令：

```
movl $0, %eax
cpuid
```

EAX中的零值定义CPUID输出选项（在这个例子中是厂商ID字符串）。CPUID运行之后，必须收集分散在3个输出寄存器中的指令响应：

```
movl $output, %edi
movl %ebx, 28(%edi)
movl %edx, 32(%edi)
movl %ecx, 36(%edi)
```

第一条指令创建一个指针，处理内存中声明的output变量时会使用这个指针。output标签的内存位置被加载到EDI寄存器中。接下来，按照EDI指针，包含厂商ID字符串片断的3个寄存器的内容被放到数据内存中的正确位置。括号外的数字表示相对于output标签的放置数据的位置。这个数字和EDI寄存器中的地址相加，确定寄存器的值被写入的地址。这个过程使用实际的厂商ID字符串片断替换用作占位符的x（注意，厂商ID字符串按照奇怪的顺序EBX、EDX和ECX分散在寄存器中）。

在内存中放置好所有厂商ID字符串片断之后，就可以显示信息了：

```
movl $4, %eax
movl $1, %ebx
movl $output, %ecx
movl $42, %edx
int $0x80
```

这个程序使用一个Linux系统调用（int $0x80）从Linux内核访问控制台显示。Linux内核提供了很多可以很容易地从汇编应用程序访问的预置函数。为了访问这些内核函数，必须使用int指令码，它生成具有0x80值的软件中断。执行的具体函数由EAX寄存器的值来确定。如果没有这个内核函数，就必须自己把每个输出字符发送到正确的显示器I/O地址。Linux系统调用为汇编语言程序员节省了大量时间。

*Linux系统调用的完整清单以及如何使用它们在第12章“使用Linux系统调用”中讨论。*

Linux的write系统调用用于把字节写入文件。下面是write系统调用的参数：

- EAX包含系统调用值。
- EBX包含要写入的文件描述符。
- ECX包含字符串的开头。
- EDX包含字符串的长度。

如果读者熟悉UNIX，就会知道几乎所有东西都被作为文件处理。标准输出（STDOUT）表示当前会话的显示终端，它的文件描述符为1。写入到这个文件描述符将在控制台屏幕上显示信息。

要显示的字节采用从之读取信息的内存位置以及要显示的字节数量的形式进行定义。ECX寄存器加载的是output标签的内存位置，它定义字符串的开头。因为输出字符串的长度总是相同的，所以可以在EDX寄存器中硬编码长度值。

显示了厂商ID信息之后，就该干净地退出程序了。同样，Linux系统调用能够提供帮助。通过使用系统调用1（退出函数），程序被正确地终止，并且返回到命令提示符。EBX寄存器包含

程序返回给shell的退出代码值。可以使用EBX寄存器的内容，按照汇编语言程序内的情况在shell脚本程序中生成不同的结果。零值表示程序成功地执行了。

### 4.2.3 构建可执行程序

有了保存为cpuid.s的汇编语言源代码程序，可以使用GNU汇编器和GNU连接器构建可执行程序，方法如下：

```
$ as -o cpuid.o cpuid.s
$ ld -o cpuid cpuid.o
$
```

这些命令的输出没有什么令人过于兴奋的（当然，除非代码中出现了一些打字错误）。第一个步骤使用as命令把汇编语言源代码汇编为目标代码文件cpuid.o。第二个步骤使用ld把目标代码文件连接为可执行文件cpuid。

如果源代码中出现了打字错误，汇编器会指出打字错误发生在哪一行中：

```
$ as -o cpuid.o cpuid.s
cpuid.s: Assembler messages:
cpuid.s:15: Error: no such instruction: 'mavl %edx,32(%edi)'
$
```

### 4.2.4 运行可执行程序

连接器生成可执行程序文件之后，就可以运行它了。下面是在我的使用奔腾4处理器的MEPIS系统上程序输出的例子：

```
$ ./cpuid
The processor Vendor ID is 'GenuineIntel'
$
```

非常好！程序按照预期运行！Linux的好处之一是，一些版本能够运行在大多数可能已经被放到一边不用的老型号计算机上。下面是老的使用Cyrix 6x86MX处理器的200MHz计算机上的输出，上面运行的系统是Mandrake Linux 6.0：

```
$ ./cpuid
The processor Vendor ID is 'CyrixInstead'
$
```

你会喜欢系统工程师的幽默的。

### 4.2.5 使用编译器进行汇编

因为GNU通用编译器（GNU Common Compiler，gcc）使用GNU汇编器编译C代码，所以也可以使用它在单一步骤内汇编和连接汇编语言程序。虽然这不是常用的方法，但是在必要的时候也是有用的。

使用gcc汇编程序时有一个问题。GNU连接器查找_start标签以便确定程序的开始位置，但是gcc查找的是main标签（读者也许已经通过C或者C++程序设计了解了它）。必须把程序中的_start

标签和定义标签的.globl命令都改成下面这样:

```
.section .text
.globl main
main:
```

改动之后，汇编和连接程序就没什么问题了:

```
$ gcc -o cpuid cpuid.s
$ ./cpuid
The processor Vendor ID is 'GenuineIntel'
$
```

## 4.3 调试程序

在这个简单的例子中，除非在源代码中出现了一些打字错误，否则程序应该正确运行并且输出期望的结果。不幸的是，在汇编语言程序设计的工作中情况不总是如此。

在更加复杂的程序中，在给寄存器和内存位置赋值或者试图使用特定指令码处理复杂数据事务时，很容易犯错误。发生错误时，可以使用便利的调试器单步运行程序并且监视数据是如何被处理的。

本节介绍如何使用GNU调试器检查范例程序，监视处理过程中寄存器和内存位置是如何改变的。

### 使用gdb

为了调试汇编语言程序，首先必须使用-gstabs参数重新汇编源代码:

```
$ as -gstabs -o cpuid.o cpuid.s
$ ld -o cpuid cpuid.o
$
```

第一次汇编源代码时，汇编过程没有发出错误或者警告消息。通过指定-gstabs参数，附加的信息被汇编进可执行程序文件中，以便帮助gdb检查源代码。虽然使用-gstabs参数创建的可执行程序的运行和行为依然和原始的程序一样，但是使用-gstabs并不是明智的做法，除非确实要调试应用程序。

因为-gstabs参数在可执行程序文件中添加了附加信息，所以产生的文件比仅仅运行应用程序所需的文件要大一些。对于这个范例程序，不使用-gstabs参数进行汇编会生成下面的文件:

```
-rwxr-xr-x   1 rich      rich          771 2004-07-13 07:32 cpuid
```

使用-gstabs参数进行汇编时，程序文件变成下面这样:

```
-rwxr-xr-x   1 rich      rich         1099 2004-07-13 07:20 cpuid
```

注意，文件长度从771字节变成了1 099字节。虽然对这个例子来说区别不大，但是想像一下如果汇编语言程序有10 000行会怎么样! 再次强调，如果并非必要，就最好不使用调试信息。

#### 1. 单步运行程序

既然可执行程序文件已经包含了必要的调试信息，可以在gdb内运行它:

```
$ gdb cpuid
GNU gdb 6.0-debian
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-linux"...
(gdb)
```

GNU调试器启动，把程序加载到内存中。可以使用run命令从gdb内运行程序:

```
(gdb) run
Starting program: /home/rich/palp/chap04/cpuid
The processor Vendor ID is 'GenuineIntel'

Program exited normally.
(gdb)
```

从输出可以看出，程序在调试器内的运行情况和从命令行运行是一样的。这并不特别令人兴奋。现在应该在程序启动时停止它，并且一步步地调试源代码的每一行。

为了进行单步调试，必须设置断点（breakpoint）。断点放置在程序代码中希望调试器停止运行程序并且查看运行情况的位置。设置断点时可以使用几个不同的选项。你可以选择在下列任何情况下停止程序的执行:

- 到达某个标签
- 到达源代码中的某个行号
- 数据值到达特定值时
- 函数执行了指定的次数之后

对于这个简单的例子，我们把断点设置在指令码的开头，并且在源代码的处理过程中监视程序的运行情况。

在汇编语言程序中指定断点时，必须指定对于最近的标签的相对位置。因为这个范例程序在指令码段中只有一个标签，所以每个断点都必须依据_start指定。break命令的格式是:

```
break * label+offset
```

其中label是被引用的源代码中的标签，offset是执行应该停止的地方距离这个标签的行数。

为了在第一条指令处设置断点，然后启动程序，应该使用下面的命令:

```
(gdb) break *_start
Breakpoint 1 at 0x8048075: file cpuid.s, line 11.
(gdb) run
Starting program: /home/rich/palp/chap04/cpuid
The processor Vendor ID is 'GenuineIntel'

Program exited normally.
(gdb)
```

这里使用*_start参数指定了断点，这个参数指定_start标签后面的第一条指令码。不幸的是，当程序运行时，它会忽略这个断点并且运行完整个程序。这是gdb当前版本中众所周知的一个缺

陷。它暂时还存在，但是希望很快能被修正。

为了解决这个问题，必须在_start标签后面的第一个指令码元素的位置包含一条伪指令。在汇编语言中，空指令称为NOP，意思是空操作。

如果修改cpuid.s源代码，在_start标签后面加上NOP指令，源代码应该是下面这个样子：

```
_start:
   nop
   movl $0, %eax
   cpuid
```

添加NOP指令之后，就可以在这个位置创建断点，表示为_start+1。现在，使用-gstabs参数进行汇编之后（不要忘记连接新的目标代码文件），可以再次试验调试器：

```
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file cpuid.s, line 12.
(gdb) run
Starting program: /home/rich/palp/chap04/cpuid

Breakpoint 1, _start () at cpuid.s:12
12          movl $0, %eax
Current language:  auto; currently asm
(gdb)
```

非常好！程序开始启动，然后暂停在（原来的）第一条指令码处。现在可以使用next或者step命令单步调试程序：

```
(gdb) next
_start () at cpuid.s:13
13          cpuid
(gdb) next
_start () at cpuid.s:14
14          movl $output, %edi
(gdb) step
_start () at cpuid.s:15
15          movl %ebx, 28(%edi)
(gdb) step
_start () at cpuid.s:16
16 @code last w/screen:movl %edx, 32(%edi)
```

每个next或者step命令会执行下一行源代码（并且显示行号是什么）。查看过感兴趣的段落之后，可以使用cont命令使程序按照正常的方式继续运行：

```
 (gdb) cont
Continuing.
The processor Vendor ID is 'GenuineIntel'

Program exited normally.
(gdb)
```

调试器从程序停止的地方继续执行并且按照正常的方式运行完程序。

虽然缓慢地单步调试程序是不错的，但是更好的是能够随着调试的进行检查数据元素。调试器为完成这样的任务提供了方法，下一节将具体讲解。

2. 查看数据

既然已经知道了如何在特定的位置停止程序，现在是在每次停止时检查数据元素的时候了。

有几个不同的gdb命令用于检查不同类型的数据元素。

两种最常被检查的数据元素是用于变量的寄存器和内存位置。下表列出了用于显示这些信息的命令。

| 数据命令 | 描　述 |
| --- | --- |
| info registers | 显示所有寄存器的值 |
| print | 显示特定寄存器或者来自程序的变量的值 |
| x | 显示特定内存位置的内容 |

使用info registers命令查看指令如何影响所有寄存器是很方便的:

```
(gdb) s
_start () at cpuid.s:13
13          cpuid
(gdb) info registers
eax            0x0        0
ecx            0x0        0
edx            0x0        0
ebx            0x0        0
esp            0xbffffd70         0xbffffd70
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x804807a          0x804807a
eflags         0x346      838
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
fs             0x0        0
gs             0x0        0
(gdb) s
_start () at cpuid.s:14
14          movl $output, %edi
(gdb) info registers
eax            0x2        2
ecx            0x6c65746e         1818588270
edx            0x49656e69         1231384169
ebx            0x756e6547         1970169159
esp            0xbffffd70         0xbffffd70
ebp            0x0        0x0
esi            0x0        0
edi            0x0        0
eip            0x804807c          0x804807c
eflags         0x346      838
cs             0x23       35
ss             0x2b       43
ds             0x2b       43
es             0x2b       43
fs             0x0        0
gs             0x0        0
(gdb)
```

输出显示，在CPUID指令执行之前，EBX、ECX和EDX寄存器都为零。CPUID指令执行之

后，它们包含从厂商ID字符串得来的值。

print命令也可以用于显示各个寄存器的值。加上一个修饰符就可以修改print命令输出格式：

- print/d显示十进制的值
- print/t显示二进制的值
- print/x显示十六进制的值

下面是print命令的一个例子：

```
(gdb) print/x $ebx
$9 = 0x756e6547
(gdb) print/x $edx
$10 = 0x49656e69
(gdb) print/x $ecx
$11 = 0x6c65746e
(gdb)
```

x命令用于显示特定内存位置的值。和print命令类似，可以使用修饰符修改x命令的输出。x命令的格式是：

```
x/nyz
```

其中n是要显示的字段数，y是输出格式，它可以是：

- c用于字符
- d用于十进制
- x用于十六进制

z是要显示的字段的长度：

- b用于字节
- h用于16位字（半字）
- w用于32位字

下面的例子使用x命令显示位于output标签的内存位置的值：

```
(gdb) x/42cb &output
0x80490ac <output>:84 'T'  104 'h' 101 'e' 32 ' '  112 'p' 114 'r' 111 'o'99 'c'
0x80490b4 <output+8>:101 'e' 115 's' 115 's' 111 'o' 114 'r' 32 ' '  86 'V' 101 'e'
0x80490bc <output+16>:110 'n' 100 'd' 111 'o' 114 'r' 32 ' ' 73 'I'  68 'D' 32 ' '
0x80490c4 <output+24>:105 'i' 115 's' 32 ' ' 39 '\'' 71 'G'  101 'e' 110 'n'117 'u'
0x80490cc <output+32>:105 'i' 110 'n' 101 'e' 73 'I' 110 'n' 116 't' 101 'e'108 'l'
0x80490d4 <output+40>:39 '\'' 10 '\n'
(gdb)
```

这个命令以字符模式（它也显示十进制值）显示output变量（&符号用于表明它是一个内存位置）的前42个字节。当跟踪对内存位置进行操作的指令时，这一特性的价值是无法衡量的。

## 4.4 在汇编语言中使用C库函数

cpuid.s程序使用Linux系统调用在控制台显示厂商ID字符串信息。还有不使用系统调用实现这个功能的其他方法。

一种方法是使用C程序员非常熟悉的标准C库函数。很容易获得这些资源来利用很多通用的

C函数。

本节介绍如何在汇编语言程序中使用C库函数。首先介绍通用的C函数printf，还有使用printf函数的cpuid.s程序的新版本。然后，下一节讲解如何汇编和连接使用C库函数的程序。

### 4.4.1 使用printf

原始的cpuid.s程序使用Linux系统调用显示程序结果。如果系统上安装了GNU C编译器，就可以很容易地使用可能早已熟悉的通用C函数。

C库包含C程序通用的很多函数，比如printf和exit。对于程序的这个版本，Linux系统调用被替换为等同的C库调用。下面是cpuid2.s程序:

```
#cpuid2.s View the CPUID Vendor ID string using C library calls
.section .data
output:
    .asciz "The processor Vendor ID is '%s'\n"
.section .bss
    .lcomm buffer, 12
.section .text
.globl _start
_start:
    movl $0, %eax
    cpuid
    movl $buffer, %edi
    movl %ebx, (%edi)
    movl %edx, 4(%edi)
    movl %ecx, 8(%edi)
    pushl $buffer
    pushl $output
    call printf
    addl $8, %esp
    pushl $0
    call exit
```

printf函数使用多个输入参数，这些参数取决于要显示的变量。第一个参数是输出字符串，带有用于显示变量的适当代码:

```
output:
    .asciz "The processor Vendor ID is '%s'\n"
```

注意，这里使用.asciz命令，而不是.ascii。printf函数要求以空字符结尾的字符串作为输出字符串。.asciz命令在定义的字符串末尾添加空字符。

使用的下一个参数是将包含厂商ID字符串的缓冲区。因为不需要定义缓冲区的值，所以在bss段中使用.lcomm命令把它声明为12个字节的缓冲区区域:

```
.section .bss
    .lcomm buffer, 12
```

CPUID指令运行之后，按照和原始的cpuid.s程序中相同的方式，包含厂商ID字符串片断的寄存器值被放到buffer变量中。

为了把参数传递给C函数printf，必须把它们压入堆栈。这是使用PUSHL指令完成的。参数放入堆栈的顺序和printf函数获取它们的顺序是相反的，所以缓冲区值被首先放入，然后是输出

字符串值。在这些操作之后，使用CALL指令调用printf函数：

```
pushl $buffer
pushl $output
call printf
addl $8, %esp
```

ADDL指令用于清空为printf函数放入堆栈的参数。使用相同的技术把零返回值放入堆栈供C函数exit使用。

### 4.4.2 连接C库函数

在汇编语言程序中使用C库函数时，必须把C库文件连接到程序目标代码。如果C库函数不可用，连接器会发生错误：

```
$ as -o cpuid2.o cpuid2.s
$ ld -o cpuid2 cpuid2.o
cpuid2.o: In function '_start':
cpuid2.o(.text+0x3f): undefined reference to 'printf'
cpuid2.o(.text+0x46): undefined reference to 'exit'
$
```

为了连接C函数库，它们在系统上必须是可用的。在Linux系统上，把C函数连接到汇编语言程序有两种方法。第一种方法称为静态连接（static linking）。静态连接把函数目标代码直接连接到应用程序的可执行程序文件中。这样会创建巨大的可执行程序，而且，如果同时运行程序的多个实例，就会造成内存浪费（每个实例都有其自己的相同函数的拷贝）。

第二种方法称为动态连接（dynamic linking）。动态连接使用库的方式使程序员可以在应用程序中引用函数，但是不把函数代码连接到可执行程序文件。在程序运行时由操作系统调用动态连接库，并且多个程序可以共享动态连接库。

在Linux系统上，标准的C动态库位于libc.so.x文件中，其中x的值代表库的版本。在我的MEPIS系统上，这个文件是libc.so.5。这个库文件包含标准C函数，包括printf和exit。

当使用gcc时，这个文件自动连接到C程序。必须手动地把它连接到程序目标代码以便C函数能够操作。为了连接libc.so文件，必须使用GNU连接器的-l参数。使用-l参数时，不需要指定完整的库名称。连接器假设库在以下文件中：

```
/lib/libx.so
```

其中x是命令行参数指定的库名称——在这个例子中是字母c。因此，连接程序的命令应该像下面这样：

```
$ ld -o cpuid2 -lc cpuid2.o
$ ./cpuid2
bash: ./cpuid2: No such file or directory
$
```

这很有趣。连接了标准C函数库文件的程序目标代码没有问题，但是当我试图运行产生的可执行文件时，出现了上述的错误消息。

问题在于连接器能够解析C函数，但是函数本身没有包含在最终的可执行程序中（记住，我

们使用的是动态连接的库）。连接器假设运行时能够找到必需的库文件。显然，在这个实例中不是这样。

为了解决这个问题，还必须指定在运行时加载动态库的程序。对于Linux系统，这个程序是ld-linux.so.2，它通常在/lib目录下。为了指定这个程序，必须使用GNU连接器的-dynamic-linker参数：

```
$ ld -dynamic-linker /lib/ld-linux.so.2 -o cpuid2 -lc cpuid2.o
$ ./cpuid2
The processor Vendor ID is 'GenuineIntel'
$
```

这样，情况好了一些。现在该运行可执行程序了，它使用ld-linux.so.2动态加载器程序查找libc.so库，程序的运行没有问题。

也可以使用gcc编译器汇编和连接汇编语言程序和C库函数。实际上，在这个例子中这要容易得多。gcc编译器自动连接必需的C库，无需进行任何特殊的操作。

首先，要记住使用gcc编译汇编语言程序时，必须把_start标签改为main。改好之后，需要做的仅仅是使用单一命令编译源代码：

```
$ gcc -o cpuid2 cpuid2.s
$ ./cpuid2
The processor Vendor ID is 'GenuineIntel'
$
```

GNU编译器自动连接正确的C库函数。

## 4.5 小结

创建汇编语言程序时，拥有所使用的汇编器的通用程序模板是个好主意。可以使用模板作为使用汇编器创建的所有程序的起点。

GNU汇编器使用的模板必须定义特定的段。GNU汇编器使用段来分隔程序中的不同数据区域。数据段包含放在特定内存位置中的由标签引用的数据。程序可以通过标签引用数据内存区域，并且在必要时修改内存位置。bss段用于包含未初始化的数据元素，比如工作缓冲区。这是创建大型缓冲区区域的理想方式。文本段用于放置实际的程序指令码。这个区域被创建之后，程序不能改动它。

模板最后的片断应该定义程序中的起始点。GNU汇编器使用_start标签声明要处理的第一条指令的位置。可以使用不同的标签，但是在连接器命令中必须使用-e参数指定这个标签。为了能够访问_start标签以便运行，还必须将它定义为全局标签。这是通过在源代码中使用.globl命令完成的。

准备好模板之后，就可以开始创建程序了。本章创建了一个简单的测试程序，它使用CPUID指令从处理器获得厂商ID字符串。该程序使用GNU汇编器汇编，使用GNU连接器连接。

程序经过测试之后，使用GNU调试器介绍如何调试汇编语言程序。必须使用-gstabs参数对程序进行汇编，以便调试器可以把指令码和源代码行匹配起来。还要记住，如果希望程序执行在第一条指令码之前停止，就要在_start标签后面马上加上NOP指令。

GNU调试器允许逐行地运行程序，并且随着程序的运行监视寄存器和内存位置的值。当试

图查找算法中的逻辑错误、或者甚至是造成指令中使用了错误的寄存器的打字错误时，这个工具的价值是无法衡量的。

最后，修改了范例程序以介绍如何在汇编语言程序中利用C函数。printf和exit函数用于显示数据和干净地退出程序。为了使用C函数，汇编语言程序必须和宿主系统上的C库连接在一起。完成这一任务的最好方法是使用C动态库。使用动态库进行连接必须要用到连接器的另一个命令行参数——-dynamic-linker参数。这指定操作系统用来动态地查找和加载库文件的程序。

对汇编语言段的介绍结束了。希望现在读者对汇编语言是什么，以及它如何有利于高级语言应用程序有了很好的理解。本书的下一部分介绍汇编语言程序设计的基础。下一章讲解有时候比较难于完成的在汇编语言程序中处理数据的任务。

# 第二部分 汇编语言程序设计基础

## 第5章 传送数据

汇编语言程序最重要的任务之一就是处理数据对象。在每个汇编语言程序中，都必须管理某种类型的数据元素。本章讨论汇编语言程序如何处理数据以及完成此任务的最佳方式。

第一节介绍在汇编语言程序中如何定义要使用的数据元素。下一节介绍如何在寄存器和内存之间传送数据。接下来，讨论条件传送指令，讲解如何依据特定操作传送数据。然后，讲解数据交换指令，介绍如何在寄存器之间以及在寄存器和内存之间交换数据。最后讨论堆栈，包括用于处理堆栈中的数据的指令。

### 5.1 定义数据元素

GNU汇编器提供了在汇编语言程序中定义和处理数据元素的很多不同方式。可以选择应用程序需要的处理数据的最佳方式。数据段和bss段都提供定义数据元素的方法。下面几节介绍汇编语言应用程序中可用的定义数据的方法。

#### 5.1.1 数据段

程序的数据段是最常见的定义数据元素的位置。数据段定义用来存储项目的特定内存位置。这些项目可以被程序中的指令码引用，并且可以被随意读取和修改。

使用.data命令声明数据段。在这个段中声明的任何数据元素都保留在内存中并且可以被汇编语言程序中的指令读取和写入。

还有另外一种类型的数据段，称为.rodata。在这种数据段中定义的任何数据元素只能按照只读（read-only）模式访问（因此使用ro前缀）。

在数据段中定义数据元素需要用到两个语句：一个标签和一个命令。

标签用作引用数据元素所使用的标记，它很像C程序中的变量名称。标签对处理器是没有意义的；它只是汇编器试图访问内存位置时用作引用指针的一个位置。

除了标签之外，还必须定义为数据元素保留多少字节。这是使用一个汇编器命令完成的。这个命令指示汇编器为通过标签引用的数据元素保留特定数量的内存。

保留的内存数量取决于定义的数据的类型，以及要声明的这个类型的项目的数量。下表介绍可以用于为特定数据元素类型保留内存的不同命令。

声明命令之后，必须定义一个（或者多个）默认值。这样把保留的内存位置中的数据设置为特定值。

| 命　令 | 数据类型 |
|---|---|
| .ascii | 文本字符串 |
| .asciz | 以空字符结尾的文本字符串 |
| .byte | 字节值 |
| .double | 双精度浮点数 |
| .float | 单精度浮点数 |
| .int | 32位整数 |
| .long | 32位整数（和.int相同） |
| .octa | 16字节整数 |
| .quad | 8字节整数 |
| .short | 16位整数 |
| .single | 单精度浮点数（和.float相同） |

下面是在数据段中声明数据元素的一个例子:

```
output:
.ascii "The processor Vendor ID is 'xxxxxxxxxxxx'\n"
```

这个代码片断留出42字节的内存，把定义的字符串顺序地存放到内存字节中，并且把标签output赋值为第一个字节。以后，当程序中引用内存位置output时，汇编器知道要转到文本字符串开头的内存位置。

相同的方法也可以用于数字。下面的代码

```
pi:
.float 3.14159
```

把3.14159的浮点表示赋值给pi标签引用的内存位置。

*在第7章“使用数字”中将更加详细地讲解如何在内存中存储浮点数。*

并不限制在一个命令语句行中只能定义一个值。可以在一行里定义多个值，按照每个值出现在命令中的顺序在内存中存放它们。例如，下面的代码

```
sizes:
.long 100,150,200,250,300
```

把长整数（4字节）值100存放在从sizes引用开始的内存位置中，然后在它后面的内存中存放4字节的值150，依此类推。这就像值的数组一样。可以通过清单中每个单独的值的相对位置引用它们。知道每个长整数值占用4个字节，可以通过访问内存位置sizes+8来访问200这个值（并且读取4个字节）。

在数据段中，可以按照需要定义多个数据元素。要记住，标签必须在定义数据的命令的前面:

```
.section .data
msg:
   .ascii "This is a test message"
factors:
    .double 37.45, 45.33, 12.30
height:
   .int 54
length:
   .int 62, 35, 47
```

按照数据段中定义数据元素的顺序，每个数据元素被存放到内存中。带有多个值的元素按照命令中列出的顺序存放。图5-1演示这种情况。

最低的内存位置包含第一个数据元素。字节被顺序地存放在内存中。下一个数据元素紧跟在前一个元素后面。

定义数据元素，然后在程序中使用它们的时候要小心。程序不会知道是否正确地处理了数据，如果你定义了两个16位整数数据值，但是把其中一个作为32位整数值引用，汇编器仍然会读取必需的4字节内存，尽管这个值是错误的。

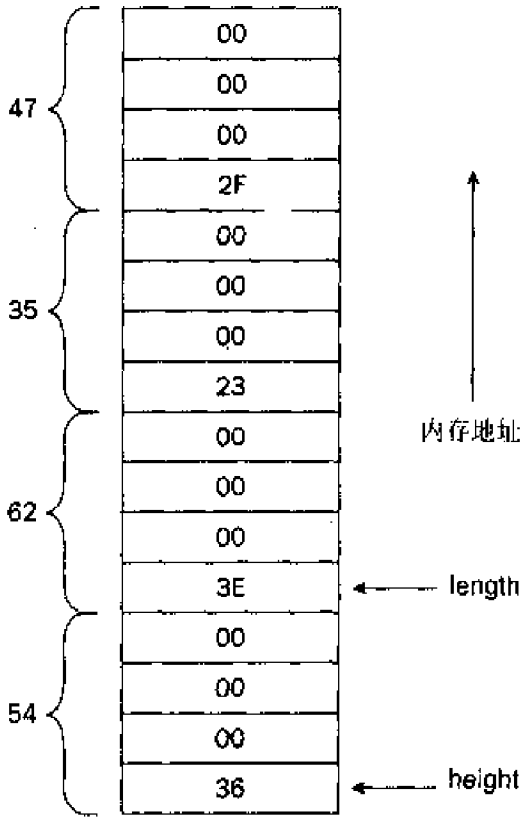



图 5-1

### 5.1.2 定义静态符号

虽然数据段主要用于定义变量数据，但是也可以在这里声明静态数据符号。.equ命令用于把常量值设置为可以在文本段中使用的符号，设置方法如下：

```
.equ factor, 3
.equ LINUX_SYS_CALL, 0x80
```

经过设置之后，数据符号值是不能在程序中改动的。.equ命令可以出现在数据段中的任何位置，但是为了使需要阅读你的程序的其他人更加方便，最好在定义其他数据之前或者之后集中定义所有数据符号。

为了引用静态数据元素，必须在标签名称前面使用美元符号（$）。例如，下面的指令

```
movl $LINUX_SYS_CALL, %eax
```

把赋值给LINUX_SYS_CALL符号的值传送给EAX寄存器。

### 5.1.3 bss段

在bss段中定义数据元素和在数据段中定义有些不同。无须声明特定的数据类型，只要声明为所需目的保留的原始内存部分即可。

GNU汇编器使用两个命令声明缓冲区，如下表所示。

| 命 令 | 描 述 |
|---|---|
| .comm | 声明未初始化的数据的通用内存区域 |
| .lcomm | 声明未初始化的数据的本地通用内存区域 |

虽然这两种区域的工作情况类似，但是本地通用内存区域是为不会从本地汇编代码之外进行访问的数据保留的。这两个命令的格式是：

```
.comm symbol, length
```

其中symbol是赋给内存区域的标签，length是内存区域中包含的字节数量，就像下面的例子所示：

```
.section .bss
.lcomm buffer, 10000
```

这些语句把10 000字节的内存区域赋值给buffer标签。在声明本地通用内存区域的程序之外的函数是不能访问它们的（不能在.globl命令中使用它们）。

在bss段中声明数据的一个好处是数据不包含在可执行程序中。在数据段中定义数据时，它必须被包含在可执行程序中，因为必须使用特定值初始化它。因为不使用程序数据初始化bss段中声明的数据区域，所以内存区域被保留在运行时使用，并且不必包含在最终的程序中。

可以通过创建用于测试的汇编语言程序，并且随着声明数据元素查看可执行文件的长度来了解这一点。首先，我们看一下没有数据元素的程序例子：

```
# sizetest1.s - A sample program to view the executable size
.section .text
.globl _start
_start:
movl $1, %eax
movl $0, %ebx
int $0x80
```

现在，汇编并且连接这个程序，然后查看它的长度：

```
$ as -o sizetest1.o sizetest1.s
$ ld -o sizetest1 sizetest1.o
$ ls -al sizetest1
-rwxr-xr-x    1 rich     rich          724 Jul 16 13:54 sizetest1*
$
```

可执行程序文件的总长度是724字节。现在，我们创建另一个测试程序，这次在bss段中声明10 000字节的缓冲区：

```
# sizetest2.s - A sample program to view the executable size
.section .bss
   .lcomm buffer, 10000
.section .text
.globl _start
_start:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

再次汇编和连接程序，然后查看它的长度：

```
$ as -o sizetest2.o sizetest2.s
$ ld -o sizetest2 sizetest2.o
$ ls -al sizetest2
-rwxr-xr-x    1 rich     rich          747 Jul 16 13:57 sizetest2*
$
```

不错。我们添加了10 000字节的缓冲区，但是可执行程序文件的长度只增加了23字节。现在，我们创建第三个测试程序，这次使用.fill命令在数据段中创建10 000字节的缓冲区：

```
# sizetest3.s - A sample program to view the executable size
```

```
.section .data
buffer:
   .fill 10000
.section .text
.globl _start
_start:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

.fill命令使汇编器自动地创建10 000个数据元素。默认为每个字段创建一个字节，并且使用零填充它。可以声明一个.byte数据值，并且自己列出10 000字节。汇编和连接这个应用程序之后，可以查看可执行程序的总长度：

```
$ as -o sizetest3.o sizetest3.s
$ ld -o sizetest3 sizetest3.o
$ ls -al sizetest3
-rwxr-xr-x    1 rich     rich        10747 Jul 16 14:00 sizetest3
$
```

看看可执行程序的长度。缓冲区空间的10 000字节被添加到了可执行程序中，使它比必要的长度大了很多。

## 5.2 传送数据元素

定义了数据元素之后，必须知道如何处理它们。因为数据元素位于内存中，并且处理器的很多指令要利用寄存器，所以处理数据元素的第一个步骤是在内存和寄存器之间传送它们。

MOV指令用作通用的数据传送指令。它是汇编语言程序中最常用的指令之一。下面几节讲解在程序中使用MOV指令传送数据的不同方式。

### 5.2.1 MOV指令格式

MOV指令的基本格式如下：

```
movx source, destination
```

source和destination的值可以是内存地址、存储在内存中的数据值、指令语句中定义的数据值，或者是寄存器。

记住，GNU汇编器使用AT&T样式的语法，所以其中的源和目标操作数的顺序和Intel文档中给出的顺序是相反的。

GNU汇编器为MOV指令添加了另一维度，在其中必须声明要传送的数据元素的长度。通过把一个附加字符添加到MOV助记符来声明这个长度。因此，指令就变成了：

```
movx
```

其中x可以是下面的字符：

- l用于32位的长字值
- w用于16位的字值
- b用于8位的字节值

因此，为了把32位的EAX寄存器值传送给32位的EBX寄存器值，你可以使用下面的指令：

```
movl %eax, %ebx
```

对于16位寄存器，指令就是：

```
movw %ax, %bx
```

对于8位寄存器：

```
movb %al, %bl
```

使用MOV指令有非常特殊的规则。只有某些位置可以传送给其他位置，MOV指令的源和目标操作数组合如下所示：

- 把立即数据元素传送给通用寄存器
- 把立即数据元素传送给内存位置
- 把通用寄存器传送给另一个通用寄存器
- 把通用寄存器传送给段寄存器
- 把段寄存器传送给通用寄存器
- 把通用寄存器传送给控制寄存器
- 把控制寄存器传送给通用寄存器
- 把通用寄存器传送给调试寄存器
- 把调试寄存器传送给通用寄存器
- 把内存位置传送给通用寄存器
- 把内存位置传送给段寄存器
- 把通用寄存器传送给内存位置
- 把段寄存器传送给内存位置

下面几节更加详细地介绍这些情况，并且给出每种情况的例子。

注意，对这些规则有一个说明。读者在第10章“处理字符串”中会发现，MOVS指令是一个作用特殊的指令，用于把字符串值从一个内存位置传送给另一个内存位置。本章中没有讨论这些指令。

### 5.2.2 把立即数传送到寄存器和内存

把数据传送给寄存器和内存位置的最容易的情况是传送立即数。立即数是在指令码语句中直接指定的，并且在运行时不能改动。

下面是传送立即数的一些例子：

```
movl $0, %eax       # moves the value 0 to the EAX register
movl $0x80, %ebx    # moves the hexadecimal value 80 to the EBX register
movl $100, height   # moves the value 100 to the height memory location
```

这些指令都在指令码中指定数据元素的值。注意，每个值前面都必须加上美元符号，表明它是立即值。立即值也可以表示为几种不同的格式——十进制（比如10、100或者230）或者十六进制（比如0x40、0x3f或者0xff）。程序被汇编和连接为可执行程序文件之后，这些值是不能被改变的。

### 5.2.3 在寄存器之间传送数据

MOV指令的下一个基本任务是把数据从一个处理器寄存器传送到另一个。这是使用处理器传送数据的最快的方式。尽可能把数据保存在处理器寄存器中通常是明智的，这样可以减少试图访问内存位置所花费的时间。

8个通用寄存器（EAX、EBX、ECX、EDX、EDI、ESI、EBP和ESP）是用于保存数据的最常用的寄存器。这些寄存器的内容可以传送给可用的任何其他类型的寄存器。和通用寄存器不同，专用寄存器（控制、调试和段寄存器）的内容只能传送给通用寄存器，或者接收从通用寄存器传送来的内容。

下面是在寄存器之间传送数据的一些例子：

```
movl %eax, %ecx  # move 32-bits of data from the EAX register to the ECX register
movw %ax, %cx    # move 16-bits of data from the AX register to the CX register
```

在长度相同的寄存器之间传送数据是很容易的。在长度不同的寄存器之间传送数据要困难一些。当指定长度大一些的寄存器接收长度小一些的数据时要小心。比如对于下面的指令：

```
movb %al, %bx
```

汇编器会报告错误。这条指令试图把AL寄存器中的8位传送给BX寄存器中的低8位。替换的做法是，应该使用MOVM指令把整个%ax寄存器的内容传送给%bx寄存器。

### 5.2.4 在内存和寄存器之间传送数据

在寄存器之间传送数据是简单的任务；不幸的是，在寄存器和内存之间传送数据就没有那么容易了。当把数据传送给内存位置，以及把内存位置中的数据传送出来的时候，必须考虑几个问题。本节讨论在内存和寄存器之间传送数据时会遇到的不同情况。

*1. 把数据值从内存传送到寄存器*

必须决定的第一件事情是如何在指令码中表示内存地址。最简单的情况是使用用于定义内存位置的标签：

```
movl value, %eax
```

这个指令把位于value标签指定的内存位置的数据值传送给EAX寄存器。实际上这个操作比看上去要困难一些。记住，MOVL指令传送32位的信息；因此，它传送从value标签引用的内存位置开始的4字节数据。如果数据长度小于4个字节，就必须使用其他MOV指令之一，比如MOVB用于1字节，或者MOVW用于2字节。

这个例子显示发生了什么：

```
# movtest1.s - An example of moving data from memory to a register
.section .data
   value:
      .int 1
.section .text
.globl _start
   _start:
      nop
      movl value, %ecx
```

```
        movl $1, %eax
        movl $0, %ebx
        int $0x80
```

现在，使用-gstabs参数汇编movtest1.s程序，连接它，并且在调试器中运行这个程序：

```
$ as -gstabs -o movtest1.o movtest1.s
$ ld -o movtest1 movtest1.o
$ gdb -q movtest1
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file movtest1.s, line 10.
(gdb) run
Starting program: /home/rich/palp/chap05/movtest1

Breakpoint 1, _start () at movtest1.s:10
10          movl (value), %ecx
Current language:  auto; currently asm
(gdb) print/x $ecx
$1 = 0x0
(gdb) next
11          movl $1, %eax
(gdb) print/x $ecx
$2 = 0x1
(gdb)
```

正如我们期望的，内存位置中存储的值被传送给了ECX寄存器。

2. 把数据值从寄存器传送给内存

使用类似的方法把数据存放回内存位置中：

```
movl %ecx, value
```

这条指令把ECX寄存器中存储的4字节数据传送给value标签指定的内存位置。就像前面那样，这条指令传送4个字节的数据，所以将使用4个内存位置存储数据。下面是使用这条指令的一段代码范例：

```
# movtest2.s - An example of moving register data to memory
.section .data
   value:
      .int 1
.section .text
.globl _start
   _start:
      nop
      movl $100, %eax
      movl %eax, value
      movl $1, %eax
      movl $0, %ebx
      int $0x80
```

同样，使用-gstabs参数汇编movtest2.s，连接它，然后在调试器中运行它：

```
$ as -o movtest2.o -gstabs movtest2.s
$ ld -o movtest2 movtest2.o
$ gdb -q movtest2
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file movtest2.s, line 11.
(gdb) run
```

```
Starting program: /home/rich/palp/chap05/movtest2

Breakpoint 1, _start () at movtest2.s:11
11          movl $100, %eax
Current language:  auto; currently asm
(gdb) x/d &value
0x804908c <value>:      1
(gdb) s
12          movl %eax, value
(gdb) s
13          movl $1, %eax
(gdb) x/d &value
0x804908c <value>:      100
(gdb)
```

通过查看value标签引用的内存位置（使用x gdb命令），会发现初始值1被存储了。单步运行程序，直到EAX寄存器的值被传送给了内存位置之后，可以再次查看该值。本次值为100，所以寄存器的值确实被存储在了内存位置中。

当访问通过标签引用的单一数据元素时，这种技术工作得很好，但是如果需要引用多个值，比如数据数组中的值，情况就变得复杂起来了。下一节介绍如何处理这些情况。

3. 使用变址的内存位置

就像前面5.1节中介绍的，可以在一个命令中指定把多个值存放到内存中：

```
values:
   .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
```

这创建存放在内存中的连续的一系列数据值。每个数据值都占用内存的一个单元（在这个例子中是长整数，即4字节）。引用数组中的数据时，必须使用变址系统确定你要访问的是哪个值。

完成这种操作的方式称为变址内存模式（indexed memory mode）。内存位置由下列因素确定：

- 基址
- 添加到基址上的偏移地址
- 数据元素的长度
- 确定选择哪个数据元素的变址

表达式的格式是：

```
base_address (offset_address, index, size)
```

获取的数据值位于：

```
base_address + offset_address + index * size
```

如果其中的任何值为零，就可以忽略它们（但是仍然需要用逗号作为占位符）。offset_address和index的值必须是寄存器，但是size的值可以是数字值。例如，为了引用前面给出的values数组中的值20，可以使用下面的指令：

```
movl $2, %edi
movl values(, %edi, 4), %eax
```

这条指令把从values标签开始的第3个4字节的变址值加载到EAX寄存器中（记住，数组从变址0开始）。大多数情况下，将使用一个寄存器计数器作为变址值，并且改变这个值来匹配要处

理的数组元素。这个过程显示在movtest3.s范例程序中:

```
# movtest3.s - Another example of using indexed memory locations
.section .data
output:
   .asciz "The value is %d\n"
values:
   .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
.section .text
.globl _start
_start:
   nop
   movl $0, %edi
loop:
movl values(, %edi, 4), %eax
pushl %eax
pushl $output
call printf
addl $8, %esp
inc %edi
cmpl $11, %edi
jne loop
movl $0, %ebx
movl $1, %eax
int $0x80
```

因为这个例子使用C函数printf，所以要记住使用所在系统上的C库动态连接器连接它:

```
$ as -o movtest3.o movtest3.s
$ ld -dynamic-linker /lib/ld-linux.so.2 -lc -o movtest3 movtest3.o
$ ./movtest3
The value is 10
The value is 15
The value is 20
The value is 25
The value is 30
The value is 35
The value is 40
The value is 45
The value is 50
The value is 55
The value is 60
$
```

程序movtest3.s遍历values标签指定的数据数组，在控制台显示其中的每个值。它使用EDI寄存器作为遍历数组用的变址:

```
movl values (, %edi, 4), %eax
```

每个值被显示之后，EDI寄存器的值被递增（使用INC指令，它把寄存器的值加1）。程序检查EDI寄存器的值，如果还没有到达最大值，就循环回去获取数组的下一个值。不必过于担心这个例子中使用的辅助代码。所有这些指令都会在以后的章节中进行讲解。现在把注意力集中在程序如何操作定义了的数据数组方面。

因为这仅仅是范例，而不是真正的应用程序，所以检查EDI寄存器是否到达数组末尾时使用

了硬编码的值。在实际的情况中，你会希望动态地确定数组中的项目数目，并且让循环进行到所有项目都被读取了的时候。

4. 使用寄存器间接寻址

除了保存数据之外，寄存器也可以用于保存内存地址。当寄存器保存内存地址时，它被称为指针（pointer）。使用指针访问存储在内存位置中的数据称为间接寻址（indirect addressing）。

这种技术可能是访问数据的操作中最容易令人困惑的部分。如果读者早已习惯于在C或者C++中使用指针，那么理解间接寻址就不会有什么问题。如果不是，那么可能需要花些时间去了解这一概念。

当使用标签引用内存位置中包含的数据值时，可以通过在指令中的标签前面加上美元符号（$）获得数据值的内存位置的地址。因此，下面这条指令

```
movl $values, %edi
```

用于把values标签引用的内存地址传送给EDI寄存器。

记住在平坦内存模型中，所有内存地址都是使用32位数字表示的。

如果阅读了第4章“汇编语言程序范例”，那么读者已经看到过间接寻址的使用。cpuid.s程序使用了下面的指令：

```
movl $output, %edi
```

这条指令把output标签的内存地址传送给EDI寄存器。标签名称前面的美元符号（$）指示汇编器使用内存地址，而不是位于这个地址的数据值。

cpuid.s程序中的下一条指令：

```
movl %ebx, (%edi)
```

是间接寻址模式的另一半。如果EDI寄存器外面没有括号，那么指令只是把EBX寄存器中的值加载到EDI寄存器中。如果EDI寄存器外面加上了括号，那么指令就把EBX寄存器中的值传送给EDI寄存器中包含的内存位置。

这是一种功能非常强大的工具。与C和C++中的指针类似，它使得可以使用寄存器来控制内存地址位置。通过递增寄存器中包含的间接寻址值，就能够体会到它真正的强大功能。不幸的是，GNU汇编器在进行这些操作时有些古怪。

GNU汇编器不允许把值与寄存器相加，必须把值放在括号之外，就像这样：

```
movl %edx, 4(%edi)
```

这条指令把EDX寄存器中的值存放在EDI寄存器指向的位置之后4个字节的内存位置中。也可以把它存放到相反的方向：

```
movl %edx, -4(%edi)
```

这条指令把值存放在EDI寄存器指向的位置之前4个字节的内存位置中。

下面是一个范例程序movtest4.s，它演示间接寻址模式：

```
# movtest4.s - An example of indirect addressing
.section .data
values:
   .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
```

```
.section .text
.globl _start
_start:
   nop
   movl values, %eax
   movl $values, %edi
   movl $100, 4(%edi)
   movl $1, %edi
   movl values(, %edi, 4), %ebx
   movl $1, %eax
   int $0x80
```

为了从这个例子获得最大的收益，使用-gstabs参数汇编它并且在调试器中监视它的运行：

```
$ gdb -q movtest4
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file movtest4.s, line 10.
(gdb) run
Starting program: /home/rich/palp/chap05/movtest4

Breakpoint 1, _start () at movtest4.s:10
10          movl values, %eax
Current language:  auto; currently asm
(gdb)
```

首先，查看values标签引用的内存位置中存储的值：

```
(gdb) x/4d &values
0x804909c <values>:     10      15      20      25
```

它加载了.int命令中指定的数据（x命令的4d参数按照十进制模式显示前4个元素）。接下来，单步运行程序，把第一个数据元素从values数组加载到EAX寄存器中：

```
(gdb) s
11          movl $values, %edi
(gdb) print $eax
$1 = 10
(gdb)
```

正如我们期望的，现在EAX寄存器包含的值为10——数组中的第一个元素。继续单步运行程序并且监视被加载到EDI寄存器中的values内存地址：

```
(gdb) s
12          movl $100, 4(%edi)
(gdb) print/x $edi
$2 = 0x804909c
(gdb)
```

现在，EDI寄存器保存十六进制值0x804909c。这是values标签引用的内存地址。下一条指令把立即数100传送到EDI寄存器指向的地址之后4字节的内存地址。这应该是values数组中的第二个数据元素。可以通过使用x命令显示的values数组看到它：

```
(gdb) s
13          movl $1, %edi
(gdb) x/4d &values
0x804909c <values>:     10      100     20      25
(gdb)
```

毫无疑问，100这个值替换了values数组中第二个数据元素值15。下一条指令把数组的第二个数据元素加载到EBX寄存器中：

```
movl $1, %edi
movl values(, %edi, 4), %ebx
```

程序的其余部分使用Linux系统调用exit终止程序。程序的退出码应该是被存放到EBX寄存器中的新创建的第二个数据数组元素（100）。可以通过在shell中查看退出码来检查这个值。这是使用特殊的环境变量$完成的：

```
$ ./movtest4
$ echo $?
100
$
```

## 5.3 条件传送指令

MOV指令是可以随意使用的功能非常强大的指令。它是大多数汇编语言程序的基础。但是，它还有可以改进之处。多年以来，Intel已经改进了IA-32平台以便提供附加的功能，使汇编语言程序员的工作更加容易。条件传送（conditional move）指令就是这些改进之一，这些指令从奔腾处理器的P6系列（奔腾Pro、奔腾II以及更新的型号）开始可用。

条件传送指令的功能就像它的名称所暗示的——MOV指令发生在特定的条件下。在旧式的汇编语言程序中，会看到下面这样的代码：

```
    dec %ecx
    jz continue
    movl $0, %ecx
continue:
```

这个代码片断首先使ECX寄存器中的值递增1。如果ECX寄存器没有溢出（进位（Carry）标志没有被设置为1），JNC指令就跳转到continue标签。如果寄存器溢出了，JNC指令就会捕获这个溢出，并且ECX寄存器被设置回0（第6章“控制执行流程”将讲解这一概念）。

ECX寄存器的值取决于它的状态，必须使用跳转指令检查它的状态。不一定非要使用跳转指令检查进位标志，可以使用条件传送指令来完成它。

虽然这是个没什么用处的例子，但是在更加复杂的产品型应用程序中，条件传送指令可以避免处理器执行JMP指令，这有助于处理器的预取缓存状态，通常能够提高应用程序的速度。

下面几节介绍条件传送指令，并且演示如何在汇编语言程序中使用它们。

### 5.3.1 CMOV指令

条件传送指令集中包含了许多指令。所有指令都具有如下格式：

```
cmovx source, destination
```

其中x是一个或者两个字母的代码，表示将触发传送操作的条件。条件取决于EFLAGS寄存器的当前值。条件传送指令使用的特定位在下表中介绍。

| EFLAGS位 | 名　称 | 描　述 |
|---|---|---|
| CF | 进位（Carry）标志 | 数学表达式产生了进位或者借位 |
| OF | 溢出（Overflow）标志 | 整数值过大或者过小 |
| PF | 奇偶校验（Parity）标志 | 寄存器包含数学操作造成的错误数据 |
| SF | 符号（Sign）标志 | 指出结果为正还是负 |
| ZF | 零（Zero）标志 | 数学操作的结果为零 |

条件传送指令成对地分组在一起，两个指令具有相同的含义。例如，一个值可以大于另一个值，但是也可以说它是不小于或者等于另一个值。这两个条件是等同的，但是二者具有各自的条件传送指令。

条件传送指令分为用于带符号操作的指令和用于无符号操作的指令。带符号操作涉及使用符号标志的比较，而无符号操作涉及忽略符号标志的比较（对带符号和无符号操作的完整介绍见第7章“使用数字”）。

下表介绍无符号条件传送指令。

| 指　令　对 | 描　述 | EFLAGS状态 |
|---|---|---|
| CMOVA/CMOVNBE | 大于/不小于或者等于 | (CF或ZF) = 0 |
| CMOVAE/CMOVNB | 大于或者等于/不小于 | CF = 0 |
| CMOVNC | 无进位 | CF = 0 |
| CMOVB/CMOVNAE | 小于/不大于或者等于 | CF = 1 |
| CMOVC | 进位 | CF = 1 |
| CMOVBE/CMOVNA | 小于或者等于/不大于 | (CF或ZF) =1 |
| CMOVE/CMOVZ | 等于/零 | ZF = 1 |
| CMOVNE/CMOVNZ | 不等于/不为零 | ZF = 0 |
| CMOVP/CMOVPE | 奇偶校验/偶校验 | PF = 1 |
| CMOVNP/CMOVPO | 非奇偶校验/奇校验 | PF = 0 |

从上表可以看出，无符号条件传送指令依靠进位、零和奇偶校验标志来确定两个操作数之间的区别。如果操作数是带符号值，就必须使用不同的条件传送指令集，如下表所示。

| 指　令　对 | 描　述 | EFLAGS状态 |
|---|---|---|
| CMOVGE/CMOVNL | 大于或者等于/不小于 | (SF异或OF) =0 |
| CMOVL/CMOVNGE | 小于/不大于或者等于 | (SF异或OF) =1 |
| CMOVLE/CMOVNG | 小于或者等于/不大于 | ((SF异或OF) 或ZF) =1 |
| CMOVO | 溢出 | OF=1 |
| CMOVNO | 未溢出 | OF=0 |
| CMOVS | 带符号（负） | SF=1 |
| CMOVNS | 无符号（非负） | SF=0 |

带符号数字和无符号数字之间的区别将在第7章中详细讨论。

带符号条件传送指令使用符号和溢出标志表示操作数之间比较的状态。

条件传送指令需要某种类型的数学指令来设置EFLAGS寄存器以便进行操作。下面是使用CMOV指令的一个例子：

```
movl value, %ecx
cmp %ebx, %ecx
cmova %ecx, %ebx
```

这个代码片断把value标签引用的数据值加载到ECX寄存器中，然后使用CMP指令把这个值和EBX寄存器中的值进行比较。CMP指令从第二个操作数中减去第一个操作数并且适当地设置EFLAGS寄存器。那么，如果ECX寄存器中的值大于EBX寄存器中的原始值，就使用CMOVA指令把EBX的值替换为ECX中的值。

记住在AT&T语法中，CMP和CMOVA指令中的操作数顺序和Intel文档中指出的顺序是相反的。这容易令人混淆。

条件传送指令使汇编语言程序员不必在比较语句之后编写跳转语句。下节中的范例程序扩展了这一概念。

### 5.3.2 使用CMOV指令

cmovtest.s程序演示条件传送指令的使用。下面是程序代码：

```
# cmovtest.s - An example of the CMOV instructions
.section .data
output:
   .asciz "The largest value is %d\n"
values:
   .int 105, 235, 61, 315, 134, 221, 53, 145, 117, 5
.section .text
.globl _start
_start:
   nop
   movl values, %ebx
   movl $1, %edi
loop:
   movl values(, %edi, 4), %eax
   cmp %ebx, %eax
   cmova %eax, %ebx
   inc %edi
   cmp $10, %edi
   jne loop
   pushl %ebx
   pushl $output
   call printf
   addl $8, %esp
   pushl $0
   call exit
```

cmovtest.s程序查找values数组中定义的一系列整数中最大的一个。它使用了一些以后的章节才介绍的指令，但是现在，只要关注条件传送语句如何工作就可以了。

EBX寄存器用于保存当前找到的最大整数。一开始，数组的第一个值被加载到EBX寄存器中。

然后数组元素被逐个地加载到EAX寄存器中，并且和EBX寄存器中的值进行比较。如果EAX寄存器中的值更大，就把它传送给EBX寄存器，并且成为新的最大值。通过在调试器中单步执行程序段，可以查看执行过程：

```
(gdb) s
14          movl values(, %edi, 4), %eax
(gdb) s
15          cmp %ebx, %eax
(gdb) print $eax
$1 = 235
(gdb) print $ebx
$2 = 105
(gdb)
```

在这里，数组中的第一个值（105）被加载到EBX寄存器中，第二个值（235）被加载到EAX寄存器中。接下来，运行CMP和CMOVA指令，并且再次检查EBX寄存器：

```
(gdb) s
16          cmova %eax, %ebx
(gdb) s
17          inc %edi
(gdb) print $ebx
$3 = 235
(gdb)
```

正如我们期望的，大一些的值（235）被传送到EBX寄存器中。这种操作一直继续，直到数组中的所有值都经过了检测。最后，保存在EBX寄存器中的值就是数组中最大的值。程序的输出应该表示出数组中的最大值。

```
$ ./cmovtest
The largest value is 315
$
```

## 5.4 交换数据

有时候在程序中必须交换数据元素的位置。MOV指令的一个缺陷是难以在不使用临时中间寄存器的情况下交换两个寄存器的值。例如，为了交换EAX和EBX寄存器中的值，必须进行类似图5-2所示的一些操作。

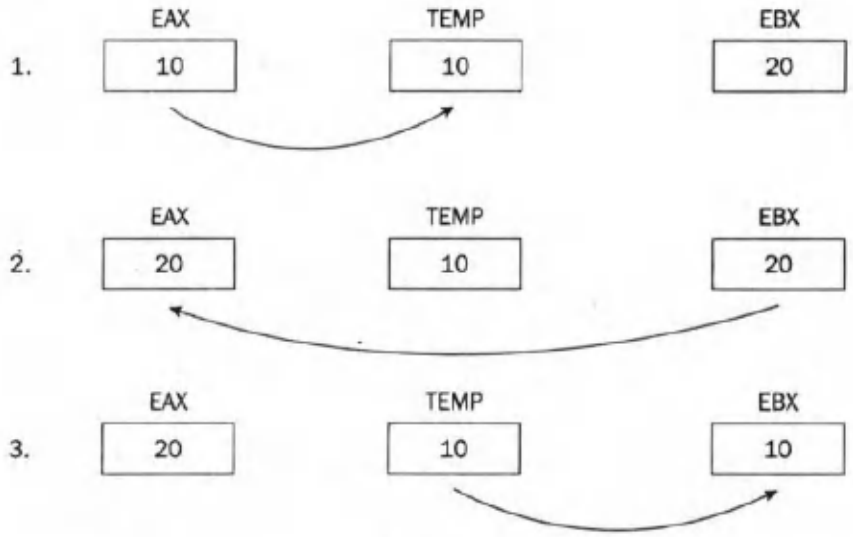



图 5-2

图5-2演示使用中间寄存器交换两个寄存器中的数据。指令码就像下面这样:

```
movl %eax, %ecx
movl %ebx, %eax
movl %ecx, %ebx
```

这需要3条指令，还要一个多余的寄存器来保存中间值。数据交换指令解决了这个问题。不使用中间寄存器，就可以在寄存器之间交换数据。本节讲解数据交换指令，并且说明如何在程序中使用它们。

### 5.4.1 数据交换指令

数据交换指令集中包含几个指令。每个指令都有特定的用途，在程序中处理数据时可以很方便地使用它们。下表介绍这些指令。

| 指 令 | 描 述 |
|---|---|
| XCHG | 在两个寄存器之间或者寄存器和内存位置之间交换值 |
| BSWAP | 反转一个32位寄存器中的字节顺序 |
| XADD | 交换两个值并且把总和存储在目标操作数中 |
| CMPXCHG | 把一个值和一个外部值进行比较，并且交换它和另一个值 |
| CMPXCHG8B | 比较两个64位值并且交换它们 |

下面几节详细地讲解每条指令。

1. XCHG

XCHG指令是这组指令中最简单的。它在两个通用寄存器之间或者寄存器和内存位置之间交换数据值。

这条指令的格式如下:

```
xchg operand1, operand2
```

operand1或者operand2可以是通用寄存器，也可以是内存位置(但是二者不能都是内存位置)。可以对任何通用的8位、16位和32位寄存器使用这个命令，但是两个操作数的长度必须相同。

当一个操作数是内存位置时，处理器的LOCK信号被自动标明，防止在交换过程中任何其他处理器访问这个内存位置。

使用XCHG对内存位置进行操作时要小心。LOCK处理是非常耗费时间的，并且可能对程序性能有不良影响。

2. BSWAP

BSWAP指令是一种功能强大的工具，当所使用的系统具有不同的字节排列方式时，它很有用。BSWAP指令反转寄存器中字节的顺序。第0～7位和第24～31位进行交换，第8～15位和第16～23位交换。图5-3演示这个操作。

记住这一点很重要：位的顺序没有被反转；被反转的是寄存器中包含的各个字节。这样就从小尾数（little-endian）的值生成了大尾数（big-endian）的值，反之亦然。

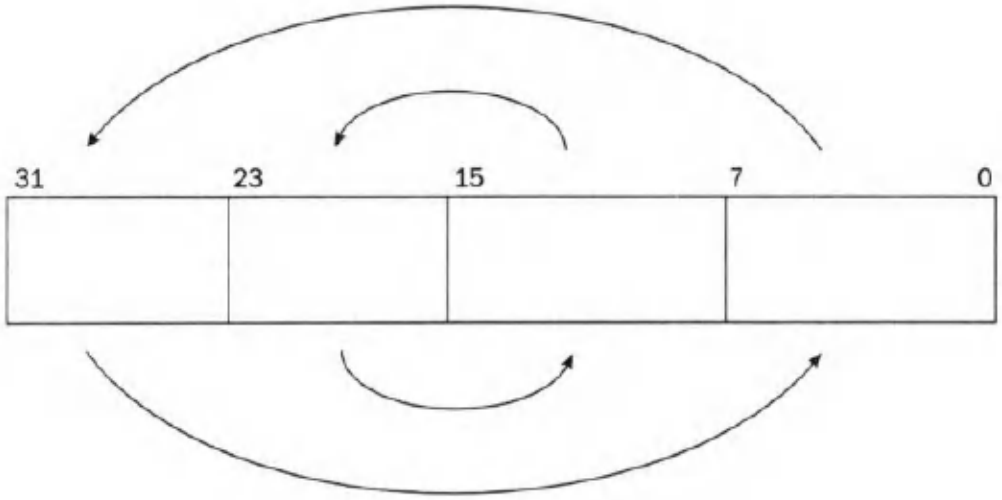



图 5-3

下面是对BSWAP指令的快速测试——swaptest.s程序:

```
# swaptest.s - An example of using the BSWAP instruction
.section .text
.globl _start
_start:
   nop
   movl $0x12345678, %ebx
   bswap %ebx
   movl $1, %eax
   int $0x80
```

这个程序简单地把十六进制值12345678加载到EBX寄存器中，然后使用BSWAP指令交换字节。可以在调试器中查看执行情况:

```
$ gdb -q swaptest
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file swaptest.s, line 5.
(gdb) run
Starting program: /home/rich/palp/chap05/swaptest

Breakpoint 1, _start () at swaptest.s:5
5          movl $0x12345678, %ebx
Current language:  auto; currently asm
(gdb) step
_start () at swaptest.s:6
6          bswap %ebx
(gdb) print/x $ebx
$1 = 0x12345678
(gdb) step
_start () at swaptest.s:7
7          movl $1, %eax
(gdb) print/x $ebx
$2 = 0x78563412
(gdb)
```

当程序在第一条MOVL指令之后停止时，可以查看EBX寄存器中的十六进制值，毫无疑问，它是12345678。现在单步执行BSWAP指令并且显示EBX寄存器的值，它是78563412，这是和原

始值相反的尾数顺序。

3. XADD

XADD指令用于交换两个寄存器或者内存位置和寄存器的值，把两个值相加，然后把结果存储在目标位置（寄存器或者内存位置）。XADD指令的格式是：

```
xadd source, destination
```

其中source必须是寄存器，destination可以是寄存器，也可以是内存位置，并且destination包含相加的结果。寄存器可以是8位、16位或者32位寄存器，XADD指令从80486处理器开始可用。

4. CMPXCHG

CMPXCHG指令比较目标操作数和EAX、AX或者AL寄存器中的值。如果两个值相等，就把源操作数的值加载到目标操作数中。如果两个值不等，就把目标操作数加载到EAX、AX或者AL寄存器中。CMPXCHG指令在早于80486之前的处理器上是不可用的。

在GNU汇编器中，CMPXCHG指令的格式是：

```
cmpxchg source, destination
```

其中的源和目标操作数的顺序和Intel文档中的顺序是相反的。目标操作数可以是8位、16位或者32位寄存器，或者是内存位置。源操作数必须是长度和目标操作数匹配的寄存器。

cmpxchgtest.s程序演示CMPXCHG指令的使用：

```
# cmpxchgtest.s - An example of the cmpxchg instruction
.section .data
data:
   .int 10
.section .text
.globl _start
_start:
   nop
   movl $10, %eax
   movl $5, %ebx
   cmpxchg %ebx, data
   movl $1, %eax
   int $0x80
```

使用CMPXCHG指令比较data标签引用的内存位置的值和EAX寄存器中的值。因为它们相等，所以源操作数（EBX）中的值被加载到data内存位置中，并且EBX寄存器的值保持不变。可以使用调试器查看程序的执行：

```
(gdb) run
Starting program: /home/rich/palp/chap05/cmpxchgtest

Breakpoint 1, _start () at cmpxchgtest.s:9
9          movl $10, %eax
Current language:  auto; currently asm
(gdb) step
10         movl $5, %ebx
(gdb) step
11         cmpxchg %ebx, data
(gdb) x/d &data
0x8049090 <data>:       10
(gdb) s
```

```
12          movl $1, %eax
(gdb) print $eax
$3 = 10
(gdb) print $ebx
$4 = 5
(gdb) x/d &data
0x8049090 <data>:       5
(gdb)
```

在CMPXCHG指令执行之前，data内存位置的值为10，它和EAX寄存器中设置的值匹配。在CMPXCHG指令执行之后，EBX中的值（这个值是5）被传送到data内存位置。

通过把赋值给data标签的值改为10以外的其他值，可以测试另一种情况。因为这个值不匹配EAX中的值，会注意到这个数据值没有改变，但是现在EAX包含在data标签中设置的值。

5. CMPXCHG8B

从CMPXCHG8B指令的名称可以猜出，它和CMPXCHG指令类似，但是有些区别——它处理8字节值（因此末尾是8B）。早于奔腾处理器的IA-32处理器不支持这条指令。CMPXCHG8B指令只有单一操作数:

```
cmpxchg8b destination
```

destination操作数引用一个内存位置，其中的8字节值会与EDX和EAX寄存器中包含的值进行比较（EDX是高位寄存器，EAX是低位寄存器）。如果目标值和EDX:EAX寄存器对中包含的值匹配，就把位于ECX:EBX寄存器对中的64位值传送给目标内存位置。如果不匹配，就把目标内存位置地址中的值加载到EDX:EAX寄存器对中。

下面的cmpxchg8btest.s程序演示这条指令:

```
# cmpxchg8btest.s - An example of the cmpxchg8b instruction
.section .data
data:
   .byte 0x11, 0x22, 0x33, 0x44, 0x55, 0x66, 0x77, 0x88

.section .text
.globl _start
_start:
   nop
   movl $0x44332211, %eax
   movl $0x88776655, %edx
   movl $0x11111111, %ebx
   movl $0x22222222, %ecx
   cmpxchg8b data
   movl $0, %ebx
   movl $1, %eax
   int $0x80
```

data标签按照定义的特定模式定义8字节内存。EAX和EDX寄存器包含相同的模式（EDX为高位字节，EAX为低位字节）。

注意和内存位置进行比较时，寄存器中的字节是如何安排的。接下来，使用完全不同的数据模式加载EBX和ECX寄存器使它们有区别。使用CMPXCHG8B指令比较data标签引用的数据和EDX:EAX寄存器对中的数据。

为了查看CMPXCHG8B指令的操作，需要在调试器中运行这个程序。首先，在

CMPXCHG8B指令执行之前查看data标签引用的数据值:

```
$ gdb -q cmpxchg8btest
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file cmpxchg8btest.s, line 10.
(gdb) run
Starting program: /home/rich/palp/chap05/cmpxchg8btest

Breakpoint 1, _start () at cmpxchg8btest.s:10
10          movl $0x44332211, %eax
Current language:  auto; currently asm
(gdb) x/8b &data
0x804909c <data>: 0x11    0x22    0x33    0x44    0x55    0x66    0x77    0x88
(gdb) s
11          movl $0x88776655, %edx
(gdb) s
12          movl $0x11111111, %ebx
(gdb) s
13          movl $0x22222222, %ecx
(gdb) s
14          cmpxchg8b data
(gdb) s
15          movl $0, %ebx
(gdb) x/8b &data
0x804909c <data>: 0x11    0x11    0x11    0x11    0x22    0x22    0x22    0x22
(gdb)
```

x命令使用8b选项显示位于data标签的全部8字节。从输出可以看出，ECX:EBX寄存器中的值确实被存放到了数据内存位置中。

### 5.4.2 使用数据交换指令

使用数据交换指令的典型范例是排序例程。有很多不同的算法用于对数据数组进行排序。其中一些算法比其他算法更加高效，但是大多数算法都搜索数据数组并且交换元素以便按照正确的顺序排列它们。

bubble.s范例使用典型的冒泡排序算法对整数数组进行排序。虽然冒泡排序法不是最高效的排序方法，但它是最容易理解和演示的。首先，我们看一下程序的源代码:

```
# bubble.s - An example of the XCHG instruction
.section .data
values:
   .int 105, 235, 61, 315, 134, 221, 53, 145, 117, 5
.section .text
.globl _start
_start:
   movl $values, %esi
   movl $9, %ecx
   movl $9, %ebx
loop:
   movl (%esi), %eax
   cmp %eax, 4(%esi)
   jge skip
   xchg %eax, 4(%esi)
   movl %eax, (%esi)
```

```
skip:
   add $4, %esi
   dec %ebx
   jnz loop
   dec %ecx
   jz end
movl $values, %esi
   movl %ecx, %ebx
   jmp loop
end:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

这是到目前为止本书中最长的程序，但是它是最有用的。再次强调，不必过于担心还没有介绍过的指令。这个程序使用了很多跳转语句，这将在第6章“控制执行流程”中详细地讲解。

从高级语言的角度来看，冒泡排序法的基本算法是这样的：

```
for(out = array_size-1; out>0, out--)
{
   for(in = 0; in < out; in++)
   {
      if (array[in] > array[in+1])
         swap(array[in], array[in+1]);
   }
}
```

这个算法有两个循环。内层循环遍历数组，检查相邻的两个数组值，找出哪个更大。如果发现大的值在小的值的前面，就交换数组中的这两个值。这个操作持续到数组的结尾。

当第一遍处理完成时，数组中的最大值应该在数组的末尾，但是其余值的顺序还没有经过排列。把具有$N$个元素的数组的所有元素排列为正确的顺序之前，必须经过$N-1$遍处理。外层循环控制总共执行了多少次内层循环。对于每遍执行内层循环，要检查的元素就少一个，因为前一次处理的最后一个元素应该已经位于正确的位置。

在汇编语言程序中，这个算法是使用一个数据数组和两个计数器（EBX和ECX）实现的。EBX计数器用于内层循环，每当检测一个数组元素时递减。当它为零时，ECX计数器递减，并且EBX计数器复位。这个过程一直持续到ECX计数器为零。这表明必须的全部处理次数都完成了。

使用间接寻址方式完成数组值的实际比较和交换。ESI寄存器加载数据数组开始位置的内存地址。然后，在比较的过程中，ESI寄存器用作指向每个数组元素的指针：

```
movl (%esi), %eax
   cmp %eax, 4(%esi)
   jge skip
   xchg %eax, 4(%esi)
   movl %eax, (%esi)
skip:
```

首先，第一个数组元素的值被加载到EAX寄存器中，然后和第二个数组元素（位于第一个元素之后4字节）的值比较。如果第二个元素已经大于或者等于第一个元素，就不做任何操作，程序继续处理下一对值。

如果第二个元素小于第一个元素，就使用XCHG指令交换第一个元素（加载到了EAX寄存

器中）和内存中的第二个元素。接下来，第二个元素（现在已经加载到了EAX寄存器中）被存放到内存中第一个元素的位置。

经过以上操作之后，ESI寄存器递增4字节，现在它指向数组中的第二个元素。然后重复以上过程，现在处理第二个和第三个数组元素。一直持续到到达数组的末尾为止。

这个简单的范例程序不生成任何输出。为了检查它是否真正工作了，可以使用调试器，并且在程序运行之前和之后查看values数组。下面是程序实际操作的输出例子：

```
$ as -gstabs -o bubble.o bubble.s
$ ld -o bubble bubble.o
$ gdb -q bubble
(gdb) break *end
Breakpoint 1 at 0x80480a5: file bubble.s, line 28.
(gdb) x/10d &values
0x80490b4 <values>:     105     235     61      315
0x80490c4 <values+16>:  134     221     53      145
0x80490d4 <values+32>:  117     5
(gdb) run
Starting program: /home/rich/palp/chap05/bubble

Breakpoint 1, end () at bubble.s:28
28          movl $1, %eax
Current language:  auto; currently asm
(gdb) x/10d &values
0x80490b4 <values>:     5       53      61      105
0x80490c4 <values+16>:  117     134     145     221
0x80490d4 <values+32>:  235     315
(gdb)
```

为了在程序的结尾捕捉数据数组的值，在end标签处创建了一个断点。程序开始时的值反映出放在.int命令定义中的值的顺序。程序运行之后，再检查这些值。毫无疑问，它们被排列成了正确的顺序。

## 5.5 堆栈

就像在第1章“什么是汇编语言”中介绍的，堆栈是程序使用的另一种内存元素。在汇编语言程序中，堆栈是最被误解的项目之一，并且给程序员新手造成了非常大的麻烦。

本节讲解堆栈和用于访问堆栈的指令。

### 5.5.1 堆栈如何工作

堆栈是内存中用于存放数据的专门保留的区域。它的特殊之处在于数据插入堆栈区域和从堆栈区域删除数据的方式。就像前面图5-1演示的，数据元素按照连续的方式存放到数据段中，在数据段中从最低的内存位置开始，向更高的内存位置依次存放。

堆栈的行为正好相反。堆栈被保留在内存区域的末尾位置，并且当数据存放在堆栈中时，它向下增长。图5-4演示这一情况。

堆栈底部（即内存顶部）包含程序运行时由操作系统存放到这里的数据元素。运行程序时使用的任何命令行参数都被送入堆栈中，并且堆栈指针被设置为指向数据元素的底部。接下来

是可以存放程序数据的区域。

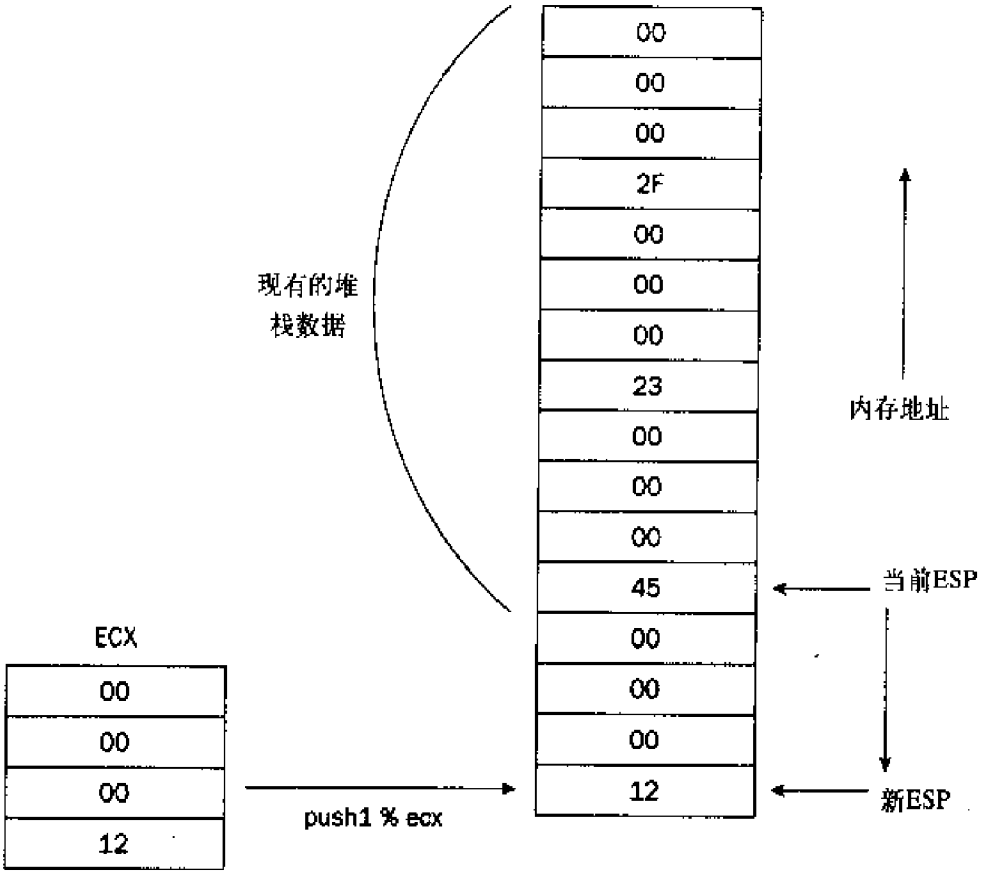



图 5-4

用来描述堆栈的典型比喻是把它看作一摞盘子。每个盘子都放在这摞盘子的最上面，以后依次从上面取盘子时会首先取它。每个数据元素都放在内存堆栈的最上面，它成为依次从堆栈获取的下一个数据元素。不可能从堆栈的中间删除数据值（虽然可以用某些技巧获得这个值）。

当每个数据被添加到堆栈区域中时，使用一个指针跟踪堆栈的开始位置在哪里。ESP寄存器包含堆栈起始位置的内存地址。虽然并未禁止，但是在程序中把ESP寄存器用作任何其他用途都是不明智的做法。如果程序丢失了堆栈的起始位置，就会发生奇怪的结果。

作为汇编语言程序员，责任是跟踪堆栈中有什么数据，并且正确地获得数据。在第11章“使用函数”中将看到，在函数之间传递数据的过程中，堆栈是重要的元素。如果在堆栈中存放了无关数据，而且没有正确地设置堆栈指针，函数就可能使用错误的值作为传递过来的参数。

不必手动地设置堆栈指针和为它操心，IA-32指令集包含一些指令帮助完成这些工作。下一节讲解用于访问堆栈中的数据的两个指令。

### 5.5.2 压入和弹出数据

把新的数据项目存到堆栈中称为压入（pushing）。用于执行这个任务的指令是PUSH指令。PUSH指令的简单格式是:

```
pushx source
```

其中x是一个字符的代码，表示数据的长度，source是要放入堆栈的数据元素。可以对其进

行PUSH操作的数据元素如下:

- 16位寄存器值
- 32位寄存器值
- 16位内存值
- 32位内存值
- 16位段寄存器
- 8位立即数值
- 16位立即数值
- 32位立即数值

用于表示数据长度的字符和MOV指令中是一样的格式，但是只能对16位和32位数据值进行PUSH操作:

- l用于长字（32位）
- w用于字（16位）

长度代码必须和指令中声明的数据元素匹配，否则会发生错误。下面是使用PUSH指令的几个例子:

```
pushl %ecx  # puts the 32-bit value of the ECX register on the stack
pushw %cx   # puts the 16-bit value of the CX register on the stack
pushl $100  # puts the value of 100 on the stack as a 32-bit integer value
pushl data  # puts the 32-bit data value referenced by the data label
pushl $data # puts the 32-bit memory address referenced by the data label
```

注意使用标签data和内存位置$data之间的区别。第一种格式（不带美元符号）把内存位置中包含的数据值存放到堆栈中，而第二种格式把标签引用的内存地址存放到堆栈中。

既然已经把所有数据放入了堆栈，就可以从堆栈获取数据了。POP指令用于完成这一部分工作。

和PUSH指令类似，POP指令使用下面的格式:

```
popx destination
```

其中x是一个字符的代码，表示数据元素的长度，destination是接收数据的位置。通过POP指令可以使用下面的数据元素接收数据:

- 16位寄存器
- 16位段寄存器
- 32位寄存器
- 16位内存位置
- 32位内存位置

显然，不能把堆栈中的数据存放到立即数值中。下面是使用POP指令的几个例子:

```
popl %ecx     # place the next 32-bits in the stack in the ECX register
popw %cx      # place the next 16-bits in the stack in the CX register
popl value  # place the next 32-bits in the stack in the value memory location
```

pushpop.s程序是把各种数据类型压入和弹出堆栈的一个例子:

```
# pushpop.s - An example of using the PUSH and POP instructions
.section .data
data:
   .int 125

.section .text
.globl _start
_start:
   nop
   movl $24420, %ecx
   movw $350, %bx
   movb $100, %eax
   pushl %ecx
   pushw %bx
   pushl %eax
   pushl data
   pushl $data

   popl %eax
   popl %eax
   popl %eax
   popw %ax
   popl %eax
   movl $0, %ebx
   movl $1, %eax
   int $0x80
```

虽然这是一个没有什么实际价值的例子，但是它能够帮助获得关于堆栈如何工作的良好概念。在调试器中运行这个程序，并且监视PUSH指令执行时ESP寄存器的值。读者应该会发现，随着每个数据元素被添加到堆栈中，ESP寄存器会递减，指向堆栈新的起始位置。这显示堆栈在内存中确实向下扩展了。

当我启动程序时，ESP寄存器包含下面的值：

```
(gdb) print/x $esp
$1 = 0xbffffd70
```

执行完所有PUSH指令之后，它包含下面的值：

```
(gdb) print/x $esp
$2 = 0xbffffd5e
```

把这两个内存位置相减，会发现ESP指针移动了18字节。把所有经过PUSH指令操作的数据相加，其总长度确实是18字节。

同样的，当POP指令从堆栈弹出每个数据元素时，ESP寄存器递增，显示堆栈长度在内存中向上缩减了。执行最后一个POP指令之后，ESP寄存器应该和原始值相等。

### 5.5.3　压入和弹出所有寄存器

下表介绍几个带来方便的附加的PUSH和POP指令。

对于同时快速地设置和获得所有通用寄存器的当前状态，PUSHA和POPA指令非常有用。PUSHA指令压入16位寄存器，使它们按照DI、SI、BP、BX、DX、CX，最后是AX的顺序出现在堆栈中。PUSHAD指令按照相同的顺序，把这些寄存器对应的32位寄存器压入堆栈。POPA和

POPAD指令按照压入寄存器的相反顺序获得寄存器状态。

| 指 令 | 描 述 |
| --- | --- |
| PUSHA/POPA | 压入或者弹出所有16位通用寄存器 |
| PUSHAD/POPAD | 压入或者弹出所有32位通用寄存器 |
| PUSHF/POPF | 压入或者弹出EFLAGS寄存器的低16位 |
| PUSHFD/POPFD | 压入或者弹出EFLAGS寄存器的全部32位 |

POPF和POPFD指令的行为因处理器的操作模式而不同。当处理器运行在保护模式下的ring 0（特权模式）下时，EFLAGS寄存器中的所有非保留标志都可以被修改，除VIP、VIF和VM标志之外。VIP和VIF标志被清零，VM标志不会被修改。

当处理器运行在保护模式的更高级别的ring（非特权模式）下时，会得到和ring 0模式下的相同结果，并且不允许修改IOFL字段。

### 5.5.4 手动使用ESP和EBP寄存器

PUSH和POP不是把数据压入和弹出堆栈的唯一途径。也可以通过使用ESP寄存器作为内存指针，手工地把数据存放到堆栈中。

通常，会看到很多程序把ESP寄存器的值复制到EBP寄存器，而不是使用ESP寄存器本身。在汇编语言函数中经常使用EBP指针指向函数的工作堆栈空间的基址。访问存储在堆栈中的参数的指令相对于EBP值引用这些参数（这将在第11章“使用函数”中详细讨论）。

## 5.6 优化内存访问

内存访问是处理器执行的最慢的功能之一。编写需要高性能的汇编语言程序时，最好尽可能地避免内存访问。只要可能，最好把变量保存在处理器的寄存器中。处理器的寄存器访问是经过高度优化的，并且是处理数据的最快方式。

当不可能把所有应用程序数据都保存在寄存器中时，应该试图优化应用程序的内存访问。对于使用数据缓存的处理器来说，在内存中按照连续的顺序访问内存能够帮助提高缓存命中率，因为内存块会一次被读取到缓存中。

当使用内存时，另一个要考虑的问题是处理器如何处理内存的读取和写入。大多数处理器（包括IA-32系列）都被优化为从数据段的开始位置，在特定的缓存块中读取和写入内存位置。在奔腾4处理器中，缓存块的长度是64位，如果定义的数据元素超过64位块的边界，就必须用两次缓存操作才能获取或者存储内存中的数据元素。

为了解决这个问题，Intel建议在定义数据时遵循下面这些原则：

- 按照16字节边界对准16位数据。
- 对准32位数据使它的基址是4的倍数。
- 对准64位数据使它的基址是8的倍数。
- 避免很多小的数据传输。而是使用单一的大型数据传输。
- 避免在堆栈中使用大的数据长度（比如80位和128位浮点值）。

在数据段中对准数据可能是困难的。数据元素被定义的顺序对于应用程序的性能可能是至关重要的。如果有很多长度类似的数据元素，比如整数和浮点值，可以把它们一起安排在数据段的开始。这确保它们保持适当的对准方式。如果有很多长度不一的数据元素，比如字符串和缓冲区，可以把它们安排在数据段的结尾，以便它们不会破坏其他数据元素的对准方式。

gas汇编器支持.align命令，它用于在特定的内存边界对准定义的数据元素。在数据段中，.align命令紧贴在数据定义的前面，它指示汇编器按照内存边界安置数据元素（这在第17章中介绍）。

## 5.7 小结

本章讨论在程序中传送数据这个非常重要的主题。几乎每个程序都必须在内存和寄存器之间传送数据元素。完成这样的工作只需要了解少数指令码即可。

在传送数据之前，必须能够在程序定义它。程序中的数据段和bss段提供可以定义数据的区域。数据段使得可以定义数据元素的默认值，比如字符串、整数和浮点数。bss段使得可以保留大量的缓冲区空间，而且无需分配默认值。默认情况下，bss段为数据的所有字节赋值为零。

MOV指令是传送数据的基本指令。它从一个寄存器到另一个寄存器、从内存位置到寄存器，或者从寄存器到内存位置传送数据。但是，它不能从一个内存位置到另一个内存位置传送数据。

MOV指令还必须包含一个结束字符，它表明数据的长度：l代表32位，w代表16位，b代表8位。两个操作数分别为源位置和目标位置（还要记住，它们的顺序和Intel格式是相反的）。

除了标准的MOV指令之外，还有条件传送指令。这些指令在特定的条件下在位置之间传送数据。CMOV指令检查进位、奇偶校验、溢出、符号和零标志以便确定应该进行传送还是不进行传送。

另一类指令是数据交换指令。这些指令可以通过单一指令交换两个独立的寄存器或者寄存器和内存位置的值。XCHG指令用于自动交换两个值。CMPXCHG和CMPXCHG8B指令首先比较目标值和特定的外部值。如果它们匹配，就进行源寄存器的交换。如果不匹配，就把目标值存放到外部位置。BSWAP指令用于交换寄存器中的高位字节和低位字节。如果需要在小尾数的计算机和大尾数的计算机之间进行通信，这个指令就很有用。

最后，讨论内存堆栈。堆栈是内存中的一个位置，一个函数或者操作可以把数据存放在这里，其他的函数或者操作可以从这里获取数据。通过这个位置可以容易地在函数之间传递数据。它也用于存放函数的本地变量。

下一章讲解改变程序的执行流程的主题。几乎所有汇编语言程序都利用某种类型的执行流程语句，按照运行过程中的变量值改变程序的行为。能够控制程序的执行流程，以及能够设计执行流程使之尽可能地平稳，是所有汇编语言程序员都需要了解的两个课题。

# 第6章 控制执行流程

在处理器运行程序时，不太可能从第一条指令开始执行，然后顺序地处理程序中的所有指令，直到最后一条指令为止。实际上，程序很可能使用分支和循环执行必需的逻辑以便实现它需要的功能。

和高级语言类似，汇编语言也提供指令来帮助程序员把逻辑编码到应用程序中。通过跳转到程序的不同段落，或者在段落内进行多次循环，可以改变程序处理数据的方式。

本章介绍用于进行跳转和循环的不同汇编语言指令。因为这两个功能都要对指令指针进行操作，所以第一节简要地复习如何使用指令指针跟踪要处理的下一条指令，以及什么指令能够改变指令指针。下一节讨论无条件分支，并且演示如何在汇编语言程序中使用它们。然后，介绍条件分支，演示如何在应用程序中使用它们实现逻辑功能。接下来的两节讲解循环，使程序能够按照预先决定的次数循环处理数据的专门指令。最后，读者将学习利用跳转和循环对优化进行优化的一些技巧。

## 6.1 指令指针

在深入研究程序执行路径的改动之前，首先要了解程序是如何在处理器上执行，这是个不错的主意。指令指针是处理器的交通警察。它确定程序中的哪条指令是应该执行的下一条指令。它按照顺序的方式处理应用程序中编写的指令码。

但是，正如第2章“IA-32平台”中讲过的，确定下一条指令在何时和何处并不总是容易的任务。随着指令预取缓存技术的发明，很多指令在实际准备好执行之前就被预先载入了处理器缓存。随着乱序引擎技术的发明，很多指令甚至在应用程序中提前执行了，其结果被安排为适当的顺序以便满足应用程序的退役单元的要求。

由于所有这些无秩序的执行方式，确定什么是确切的“下一条指令”可能是困难的。虽然有很多工作在幕后进行，用以提高程序的执行速度，但是处理器仍然需要顺序地单步执行程序逻辑以便生成正确的结果。在这个框架之内，指令指针对于确定程序中执行到了什么位置是至关重要的。这显示在图6-1中。

当处理器退役单元执行来自指令的乱序引擎的结果时，指令就被认为执行过了。指令执行之后，指令指针递增到程序代码中的下一条指令。这条指令或许已经由乱序引擎执行过了，或许还没有执行，但是不管是哪一种情况，退役单元都不会处理其结果，直到程序逻辑中应该这样做的时候。

当指令指针在程序指令中移动时，EIP寄存器会递增。记住，指令的长度可能是多个字节，所以指向下一条指令不仅仅是每次使指令指针递增1。

程序不能直接修改指令指针。程序员不具有使用MOV指令直接将EIP寄存器的值改为指向内存中的不同位置的能力。但是，可以利用能够改动指令指针值的指令。这些指令称为分支（branch）。

分支指令可以改动EIP寄存器的值，要么是无条件改动（无条件分支），要么是按照条件值改动（条件分支）。下面几节讲解无条件分支和条件分支，并且介绍它们是如何影响指令指针和程序逻辑的执行路径的。

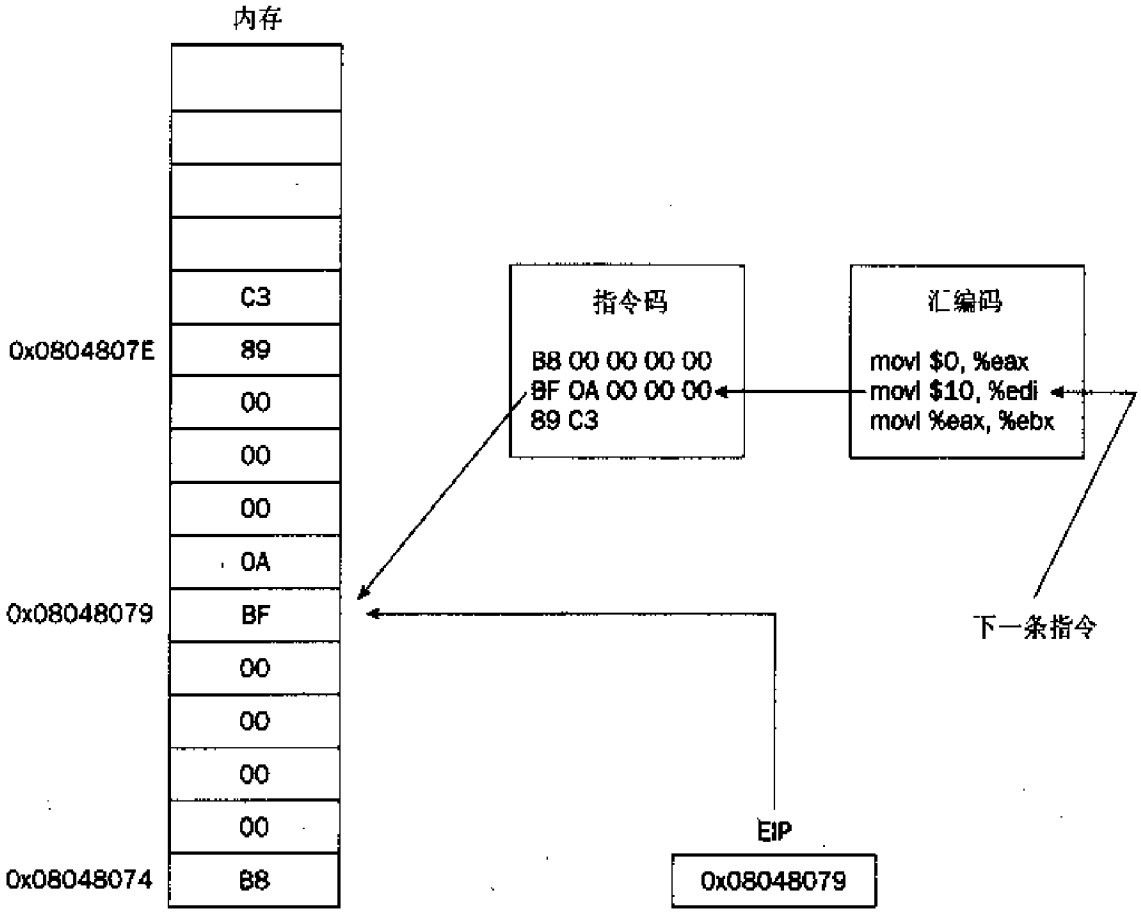



图 6-1

## 6.2 无条件分支

程序中遇到无条件分支时，指令指针自动转到另一个位置。可以使用的无条件分支有3种：

- 跳转
- 调用
- 中断

每种无条件分支在程序中的行为都不同，可以决定在程序逻辑中使用哪一种。下面几节讨论这些种类的无条件分支之间的区别，以及如何在汇编语言程序中实现它们。

### 6.2.1 跳转

跳转是汇编语言程序设计中最基本的分支类型。如果熟悉BASIC程序设计语言，读者很可能见过GOTO语句。汇编语言的跳转语句和BASIC语言的GOTO语句是等同的。

在结构化程序设计中，GOTO被认为是不良编码的标志。程序被划分为几个区域并且按照顺序的流程方式执行，调用函数，而不是在程序代码中跳转。在汇编语言程序中，不认为跳转指

令是不良的程序设计，而且实际上必须使用它实现很多功能。但是，它们对程序的性能有负面的影响（参见本章后面的6.6节）。

跳转指令使用单一指令码:

```
jmp location
```

其中location是要跳转到的内存地址。在汇编语言中，这个位置值被声明为程序代码中的标签。遇到跳转时，指令指针改变为紧跟在标签后面的指令码的内存地址。图6-2演示这种情况。

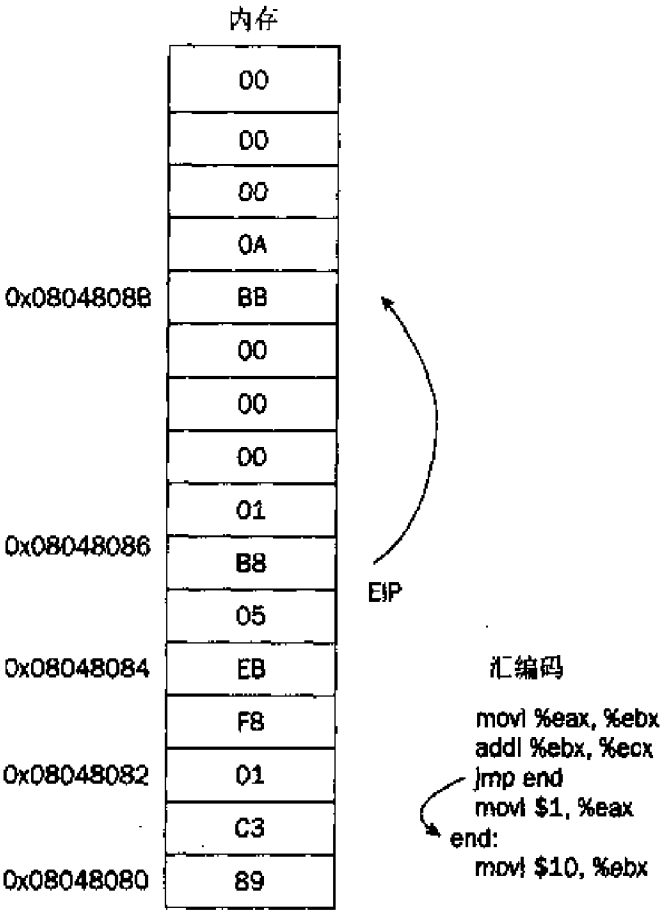



图 6-2

JMP指令把指令指针的值改变为JMP指令中指定的内存位置。

在幕后，单一汇编跳转指令被汇编为跳转操作码的3种不同类型之一:

- 短跳转
- 近跳转
- 远跳转

这3种跳转类型是由当前指令的内存位置和目的点（“跳转到”的位置）的内存位置之间的距离决定的。依据跳过的字节数目决定使用哪种跳转类型。当跳转偏移量小于128字节时使用短跳转。在分段内存模式下，当跳转到另一个段中的指令时使用远跳转。近跳转用于所有其他跳转。

使用汇编语言助记符指令时，不需要担心跳转的长度。单一跳转指令用于跳转到程序代码中的任何位置（虽然可能有性能上的差别，后面6.6节讨论这个问题）。

下面的jumptest.s程序演示无条件跳转指令的操作:

```
# jumptest.s - An example of the jmp instruction
.section .text
.globl _start
_start:
   nop
   movl $1, %eax
   jmp overhere
   movl $10, %ebx
   int $0x80
overhere:
   movl $20, %ebx
   int $0x80
```

jumptest.s程序简单地把EAX寄存器赋值为1，用于进行Linux系统调用exit。然后，使用跳转指令跳过把EBX寄存器赋值为10，并且进行Linux系统调用的部分。程序跳转到把EBX寄存器赋值为20、并且进行Linux系统调用的位置。通过在Linux环境中运行程序并且查看生成的结果代码，可以确定跳转发生了：

```
$ as -o jumptest.o jumptest.s
$ ld -o jumptest jumptest.o
$ ./jumptest
$ echo $?
20
$
```

确实，期望的结果代码是由跳转生成的。这本身可能不太令人兴奋。换种方式，可以使用调试器和objdump程序监视程序中使用的实际内存位置。

首先，通过使用objdump程序反汇编经过汇编的代码，可以了解指令码是如何在内存中安排的：

```
$ objdump -D jumptest

jumptest:     file format elf32-i386

Disassembly of section .text:

08048074 <_start>:
 8048074:       90                      nop
 8048075:       b8 01 00 00 00          mov    $0x1,%eax
 804807a:       eb 07                   jmp    8048083 <overhere>
 804807c:       bb 0a 00 00 00          mov    $0xa,%ebx
 8048081:       cd 80                   int    $0x80

08048083 <overhere>:
 8048083:       bb 14 00 00 00          mov    $0x14,%ebx
 8048088:       cd 80                   int    $0x80
$
```

反汇编器的输出显示每条指令将使用的内存位置（值显示在第一列中）。现在，你可以在调试器中运行jumptest并且监视程序的运行情况：

```
$ as -gstabs -o jumptest.o jumptest.s
$ ld -o jumptest jumptest.o
$ gdb -q jumptest
(gdb) break *_start+1
```

```
Breakpoint 1 at 0x8048075: file jumptest.s, line 5.
(gdb) run
Starting program: /home/rich/palp/chap06/jumptest

Breakpoint 1, _start () at jumptest.s:5
5           movl $1, %eax
Current language:  auto; currently asm
(gdb) print/x $eip
$1 = 0x8048075
(gdb)
```

在调试器对代码进行汇编（使用-gstabs参数），并且在程序的开始位置设置断点之后，运行程序并且查看使用的第一个内存位置（显示在EIP寄存器中）。这个值是0x8048075，它和objdump输出中显示的相同内存位置相对应。下一步，单步运行调试器，直到执行过跳转指令，然后再次显示EIP寄存器的值：

```
(gdb) step
_start () at jumptest.s:6
6           jmp overhere
(gdb) step
overhere () at jumptest.s:10
10          movl $20, %ebx
(gdb) print $eip
$2 = (void *) 0x8048083
(gdb)
```

正如我们期望的，程序跳转到0x8048083的位置，这正是overhere标签指向的位置，就像objdump输出显示的一样。

### 6.2.2 调用

无条件分支的下一种类型是调用。调用和跳转指令类似，但是它保存发生跳转的位置，并且它具有在需要的时候返回这个位置的能力。在汇编语言程序中实现函数时使用它。

用函数可以编写划分为区域的代码，可以把不同功能分隔为不同的文本段落。如果程序的多个区域使用相同的函数，那么就不需要多次编写相同的代码。可以使用调用语句引用单一函数。

调用指令有两个部分。第一个部分是实际的CALL指令，它需要单一操作数——跳转到的位置的地址：

```
call address
```

address操作数引用程序中的标签，它被转换为函数中的第一条指令的内存地址。

调用指令的第二部分是返回指令。它使函数可以返回代码的原始部分，就是紧跟在CALL指令后面的位置。返回指令没有操作数，只有助记符RET。通过查看堆栈，它知道应该返回到什么位置。图6-3演示这些情况。

执行CALL指令时，它把EIP寄存器的值存放到堆栈中，然后修改EIP寄存器以指向被调用的函数地址。当被调用的函数完成后，它从堆栈获得过去的EIP寄存器值，并且把控制权返回给原始程序。

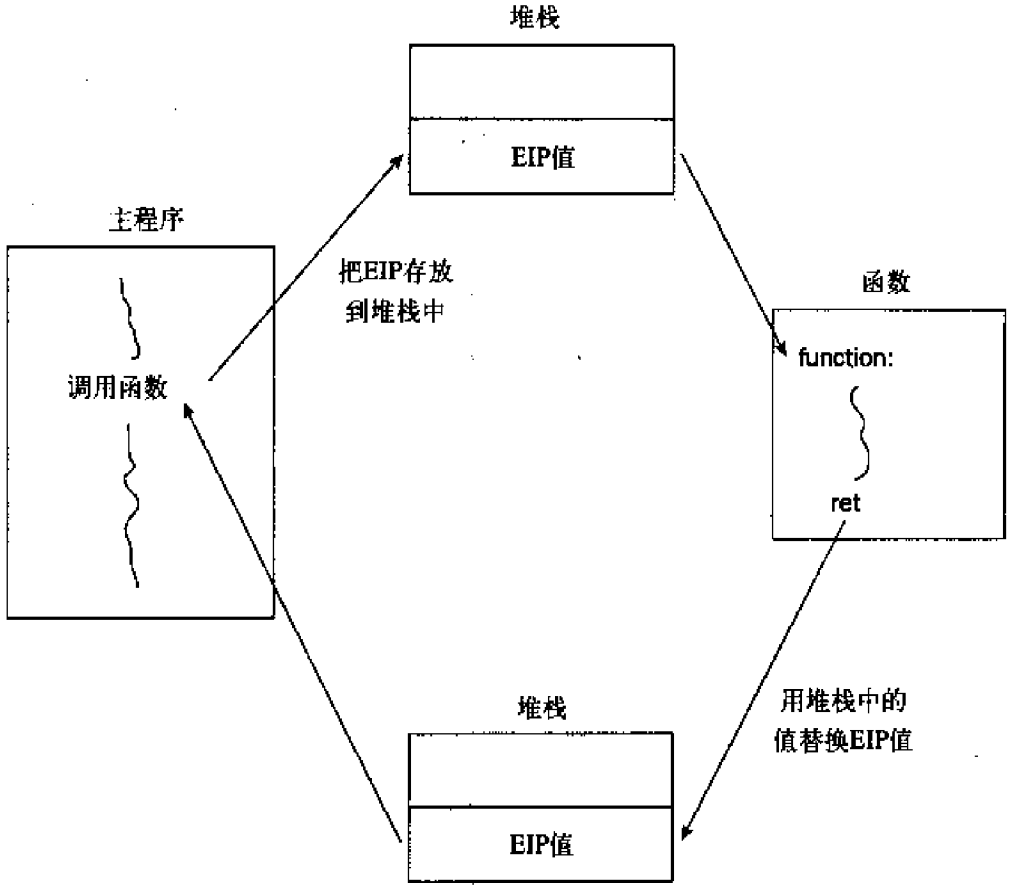



图 6-3

函数如何返回主程序可能是在汇编语言中使用函数时最容易混淆的部分。这个过程不仅仅是在函数的结尾使用RET指令那么简单。实际上，这关系到如何把信息传递给函数以及函数如何读取和存储这些信息。

这些操作是使用堆栈完成的。就像第5章中讲过的，不仅可以使用PUSH和POP指令引用堆栈中的数据，而且可以使用ESP寄存器直接引用数据，ESP寄存器指向堆栈中的最后一个条目。函数通常把ESP寄存器复制到EBP寄存器，然后使用EBP寄存器值获得CALL指令之前传递给堆栈的信息，并且为本地数据存储把变量存放在堆栈中（参见第11章）。这使如何在函数之内操作堆栈指针变得复杂起来。

*这是关于函数如何使用堆栈的稍微经过简化的解释。第11章将详细地进行讲解，介绍和演示函数如何为数据存储使用堆栈。*

当执行CALL指令时，返回地址被添加到堆栈中。当被调用的函数开始时，它必须在某个位置存储ESP寄存器，在RET指令试图返回发出调用的程序之前，被调用函数可以从这个存储位置恢复ESP寄存器的原始形式。因为在函数中也可能对堆栈进行操作，所以EBP经常用作堆栈的基指针。因此，当函数的开始时，通常也把ESP寄存器复制到EBP寄存器。

虽然看上去有些混乱，但是如果创建用于所有函数调用的标准模板，这些操作并不太困难。下面是用于函数的模板形式：

```
function_label:
    pushl %ebp
```

```
movl %esp, %ebp
< normal function code goes here>
movl %ebp, %esp
popl %ebp
ret
```

保存了EBP寄存器之后，就可以使用它作为堆栈的基指针，以便在函数中进行对堆栈的所有访问。在返回发出调用的程序之前，ESP寄存器必须被恢复为指向发出调用的内存位置。

calltest.s程序中演示一个简单的调用例子：

```
# calltest.s - An example of using the CALL instruction
.section .data
output:
   .asciz "This is section %d\n"
.section .text
.globl _start
_start:
   pushl $1
   pushl $output
   call printf
   add  $8, %esp            # should clear up stack
   call overhere
   pushl $3
   pushl $output
   call printf
   add  $8, %esp            # should clear up stack
   pushl $0
   call exit
overhere:
   pushl %ebp
   movl %esp, %ebp
   pushl $2
   pushl $output
   call printf
   add  $8, %esp            # should clear up stack
   movl %ebp, %esp
   popl %ebp
   ret
```

在calltest.s程序的开始，使用C函数printf显示第一个文本行，显示程序处于什么位置。下一步，使用CALL指令把控制转移到overhere标签。在overhere标签，ESP寄存器被复制给EBP指针，以便在函数的结尾可以恢复它。再次使用printf函数显示第二行文本，然后恢复ESP和EBP寄存器。

程序的控制返回到紧跟在CALL指令后面的指令，并且再次使用printf函数显示第三个文本行。输出应该如下：

```
$ ./calltest
This is section 1
This is section 2
This is section 3
$
```

当从应用程序调用外部函数时要小心，不能保证函数返回寄存器的方式和发出调用之前离开寄存器的方式相同。第11章中将更加详细地讨论这个主题。

### 6.2.3 中断

无条件分支的第三种类型是中断。中断是处理器“中断”当前指令码路径并且切换到不同路径的方式。中断有两种形式:

- 软件中断
- 硬件中断

硬件设备生成硬件中断。使用硬件中断发出信号，表示硬件层发生的事件（比如I/O端口接收到输入信号时）。程序生成软件中断。它们是把控制交给另一个程序的信号。

当一个程序被中断调用时，发出调用的程序暂停，被调用的程序接替它运行。指令指针被转移到被调用的程序，并且从被调用的程序内继续执行。被调用的程序完成时，它可以把控制返回给发出调用的程序（使用中断返回指令）。

软件中断是操作系统提供的，使应用程序可以使用操作系统内的函数，并且，在某些情况下，甚至可以接触底层的BIOS系统。在Microsoft DOS操作系统中，为很多函数提供了0x21软件中断。在Linux领域，0x80中断用于提供低级内核函数。

到目前为止，在本书提供的很多范例程序中，已经看到过了使用软件中断的几个例子。简单地使用带有0x80值的INT指令把控制转移给Linux系统调用程序。Linux系统调用程序具有很多可以使用的子函数。中断发生时，按照EAX寄存器的值执行子函数。例如，在中断调用Linux系统调用函数exit之前，把值1存放到EAX寄存器中。

第12章讲解所有通过0x80中断可用的函数。

调试包含软件中断的应用程序时，查看中断段落之内发生了什么很困难，因为调试信息没有被编译到函数里面。读者也许已经注意到运行调试器时，它执行中断指令，但是马上返回到一般程序。

## 6.3 条件分支

和无条件分支不同，条件分支不总是被执行。条件分支的结果取决于执行分支时EFLAGS寄存器的状态。

EFLAGS寄存器中有很多位，但是条件分支只和其中的5位有关:

- 进位（Carry）标志（CF）——第0位（借位有效位）
- 溢出（Overflow）标志（OF）——第11位
- 奇偶校验（Parity）标志（PF）——第2位
- 符号（Sign）标志（SF）——第7位
- 零（Zero）标志（ZF）——第6位

每个条件跳转指令都检查特定的标志位以便确定是否符合进行跳转的条件。使用这5个不同的标志位，可以执行几种跳转组合。下面几节介绍各个跳转指令。

### 6.3.1 条件跳转指令

条件跳转按照EFLAGS寄存器的当前值来确定是否进行跳转。几种不同的条件跳转指令使用

EFLAGS寄存器的不同位。条件跳转指令的格式如下:

```
jxx address
```

其中xx是1个到3个字符的条件代码，address是程序要跳转到的位置（通常以标签表示）。下表介绍所有可用的条件跳转指令。

| 指 令 | 描 述 | EFLAGS |
|---|---|---|
| JA | 如果大于（above），则跳转 | CF=0与ZF=0 |
| JAE | 如果大于（above）或等于，则跳转 | CF=0 |
| JB | 如果小于（below），则跳转 | CF=1 |
| JBE | 如果小于（below）或等于，则跳转 | CF=1或ZF=1 |
| JC | 如果进位，则跳转 | CF=1 |
| JCXZ | 如果CX寄存器为0，则跳转 | |
| JECXZ | 如果ECX寄存器为0，则跳转 | |
| JE | 如果相等，则跳转 | ZF=1 |
| JG | 如果大于（greater），则跳转 | ZF=0与SF=OF |
| JGE | 如果大于（greater）或等于，则跳转 | SF=OF |
| JL | 如果小于（less），则跳转 | SF<>OF |
| JLE | 如果小于（less）或等于，则跳转 | ZF=1或SF<>OF |
| JNA | 如果不大于（above），则跳转 | CF=1或ZF=1 |
| JNAE | 如果不大于（above）或等于，则跳转 | CF=1 |
| JNB | 如果不小于（below），则跳转 | CF=0 |
| JNBE | 如果不小于（below）或等于，则跳转 | CF=0与ZF=0 |
| JNC | 如果无进位，则跳转 | CF=0 |
| JNE | 如果不等于，则跳转 | ZF=0 |
| JNG | 如果不大于（greater），则跳转 | ZF=1或SF<>OF |
| JNGE | 如果不大于（greater）或等于，则跳转 | SF<>OF |
| JNL | 如果不小于（less），则跳转 | SF=OF |
| JNLE | 如果不小于（less）或等于，则跳转 | ZF=0与SF=OF |
| JNO | 如果不溢出，则跳转 | OF=0 |
| JNP | 如果不奇偶校验，则跳转 | PF=0 |
| JNS | 如果无符号，则跳转 | SF=0 |
| JNZ | 如果非零，则跳转 | ZF=0 |
| JO | 如果溢出，则跳转 | OF=1 |
| JP | 如果奇偶校验，则跳转 | PF=1 |
| JPE | 如果偶校验，则跳转 | PF=1 |
| JPO | 如果奇校验，则跳转 | PF=0 |
| JS | 如果带符号，则跳转 | SF=1 |
| JZ | 如果为零，则跳转 | ZF=1 |

读者也许注意到很多条件跳转指令似乎是多余的（比如，如果是above大于，则使用JA进行跳转；如果是greater大于，则使用JG进行跳转）。不同之处在于处理带符号值和无符号值的时候。对于计算无符号整数值，跳转指令使用above和below关键字。对于带符号整数值，使用greater和less。在第7章“使用数字”中，将学习更多关于不同整数类型的知识。

在指令码中，条件跳转指令使用单一操作数——要跳转到的地址。这个操作数常常是汇编

语言程序中的一个标签，而在指令码中被转换为偏移地址。条件跳转允许两种跳转类型:

- 短跳转
- 近跳转

短跳转使用8位带符号地址偏移量，而近跳转使用16位或者32位带符号地址偏移量。偏移量值被加到指令指针上。

条件跳转指令不支持分段内存模式下的远跳转。如果在分段内存模式下进行程序设计，就必须使用程序设计逻辑确定条件是否存在，然后实现无条件跳转转移到另一个段中的指令。

为了能够使用条件跳转，在进行跳转之前，必须进行设置EFLAGS寄存器的操作。下一节介绍在汇编语言程序中使用条件跳转的几个例子。

### 6.3.2 比较指令

比较指令是为进行条件跳转而比较两个值的最常见的途径。比较指令的作用就像它的名称表示的，它比较两个值并且相应地设置EFLAGS寄存器。

CMP指令的格式如下:

```
cmp operand1, operand2
```

CMP指令把第二个操作数和第一个操作数进行比较。在幕后，它对两个操作数执行减法操作（operand2-operand1）。比较指令不会修改这两个操作数，但是如果发生减法操作，就设置EFLAGS寄存器。

使用GNU汇编器时，记住在CMP指令中，operand1和operand2的顺序与Intel文档中的顺序是相反的。这个微小的差异使很多汇编语言程序员花费了很多时间调试它导致的错误。

下面的cmptest.s程序演示如何一起使用比较和条件跳转指令:

```
# cmptest.s - An example of using the CMP and JGE instructions
.section .text
.globl _start
_start:
   nop
   movl $15, %eax
   movl $10, %ebx
   cmp %eax, %ebx
   jge greater
   movl $1, %eax
   int $0x80
greater:
   movl $20, %ebx
   movl $1, %eax
   int $0x80
```

cmptest.s首先赋值立即值：把15赋值给EAX寄存器，把10赋值给EBX寄存器。下一步，使用CMP指令比较这两个寄存器，然后按照比较的结果，使用JGE指令进行分支操作:

```
cmp %eax, %ebx
jge greater
```

因为EBX寄存器的值小于EAX的值，所以没有执行条件分支。指令指针转向下一条指令，它把立即值1存放到EAX寄存器中，然后调用Linux系统调用exit。可以运行程序并且显示结果代

码来检验此过程:

```
$ ./cmptest
$ echo $?
10
$
```

确实，没有进行分支操作，EBX寄存器中的值仍然是10。

前面的例子比较两个寄存器的值。下面是CMP指令的一些其他例子:

```
cmp $20, %ebx      ; compare EBX with the immediate value 20
cmp data, %ebx     ; compare EBX with the value in the data memory location
cmp (%edi), %ebx ; compare EBX with the value referenced by the EDI pointer
```

### 6.3.3 使用标志位的范例

试图编写条件跳转指令可能比较需要技巧。了解每个标志位对于满足不同条件所需的状态对编写程序是有帮助的。下面几节演示每个标志位如何影响条件跳转，以便读者可以了解编写程序逻辑时应该注意什么问题。

1. 使用零标志

进行条件跳转时，零标志的使用是最简单的。如果零标志被置1（两个操作数相等），JE和JZ指令就跳转到分支。零标志可以由CMP指令设置，也可以由计算结果为零的数学指令设置，如下面的例子所示:

```
movl $30, %eax
subl $30, %eax
jz overthere
```

JZ指令将被执行，因为SUB指令的结果为零（SUB指令将在第8章“基本数学函数”中讲解）。

也可以在递减寄存器的值时使用零标志，以便确定它是否到达零:

```
movl $10, %edi
loop1:
   < other code instructions>
   dec %edi
   jz out
   jmp loop1
out:
```

这个代码片断使用EDI寄存器作为变址计数器，它从10递减到1（当它到达零时，JZ指令将退出循环）。

2. 使用溢出标志

溢出标志专门用在处理带符号数字时（参见第7章）。当带符号值对于包含它的数据元素来说太大时，溢出标志被设置为1。这经常发生在溢出了保存数据的寄存器长度的数学操作的过程中，如下面的例子所示:

```
movl $1, %eax        ; move 1 to the EAX register
movb $0x7f, %bl      ; move the signed value 127 to the 8-bit BL register
addb $10, %bl        ; Add 10 to the BL register
jo overhere
int $0x80            ; call the Linux system call
```

```
overhere:
   movl $0, %ebx        ; move 0 to the EBX register
   int $0x80            ; call the Linux system call
```

这个代码片断把带符号字节值127加上10。结果应该是137，这对于字节来说是合法值，但是对带符号字节数是非法的（带符号字节数只能使用-127到127的值）。因为这个带符号值非法，所以设置溢出标志为1，并且执行JO指令。

3. 使用奇偶校验标志

奇偶校验标志表明数学运算答案中应该为1的位的数目。可以使用它作为粗略的错误检查系统，确保数学操作成功执行。

如果结果中被设置为1的位的数目是偶数，则设置奇偶校验位（置1）。如果设置为1的位的数目是奇数，则不设置奇偶校验位（置0）。

为了测试这个概念，可以创建paritytest.s程序：

```
# paritytest.s - An example of testing the parity flag
.section .text
.globl _start
_start:
   movl $1, %eax
   movl $4, %ebx
   subl $3,  %ebx
   jp overhere
   int $0x80
overhere:
   movl $100, %ebx
   int $0x80
```

在这个代码片断中，减法的结果是1，以二进制表示是00000001。因为为1的位的数目是奇数，所以不设置奇偶校验位，JP指令不会跳转到分支；程序退出，并且以减法的结果1作为结果代码：

```
$ ./paritytest
$ echo $?
1
$
```

为了测试相反的情况，改变SUB指令这一行，使生成的结果中为1的位的数目为偶数：

```
subl $1, %ebx
```

这次减法的结果是3，以二进制表示是00000011。因为为1的位的数目是偶数，所以设置奇偶校验位，并且JP指令应该转到overhere标签的分支，设置结果代码为100：

```
$ ./paritytest
$ echo $?
100
$
```

正是我们期望的结果！

4. 使用符号标志

符号标志使用在带符号数中，用于表示寄存器中包含的值的符号改变。在带符号数中，最

后一位（最高位）用作符号位。它表明数字表示是负值（设置为1）还是正值（设置为0）。

当在循环内进行计数并且监视零值时，这个标志很有用。我们讲解过零标志，当值被递减到达零时设置零标志。但是，如果正在处理数组，很可能也需要在零值之后才停止，而不是在到达零值的位置停止（因为第一个偏移量是0）。

使用符号标志，可以得到值变从0到-1的通知，如下面的signtest.s程序所示：

```
# signtest.s - An example of using the sign flag
.section .data
value:
   .int 21, 15, 34, 11, 6, 50, 32, 80, 10, 2
output:
   .asciz "The value is: %d\n"
.section .text
.globl _start
_start:
   movl $9, %edi
loop:
   pushl value(, %edi, 4)
   pushl $output
   call printf
   add $8, $esp
   dec %edi
   jns loop
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

signtest.s程序反向遍历数据数组，使用EDI寄存器作为变址，处理每个数组元素时递减这个寄存器。使用JNS指令检查EDI寄存器的值什么时候变成负值，如果不是负值，则返回到循环的开头。

因为signtest.s程序使用C函数printf，所以要记住用动态加载器连接它（这在第4章“汇编语言程序范例”中讲解过）。程序的输出应该如下：

```
$ ./signtest
The value is: 2
The value is: 10
The value is: 80
The value is: 32
The value is: 50
The value is: 6
The value is: 11
The value is: 34
The value is: 15
The value is: 21
$
```

5. 使用进位标志

进位标志用在数学表达式中，表示无符号数中何时发生溢出（记住带符号数使用溢出标志）。当指令导致寄存器超出其数据长度限制时设置进位标志。

和溢出标志不同，DEC和INC指令不影响进位标志。例如，下面这个代码片断不会设置进位标志：

```
movl $0xffffffff, %ebx
```

```
inc %ebx
jc overflow
```

但是，下面这个代码片断会设置进位标志，并且JC指令会跳转到overflow的位置：

```
movl $0xffffffff, %ebx
addl $1,  %ebx
jc overflow
```

当无符号值小于零时也会设置进位标志。例如，下面这个代码片断也会设置进位标志：

```
movl $2, %eax
subl $4, %eax
jc overflow
```

EAX寄存器中的结果值是254，作为带符号数，它代表-2，即正确的答案。这就是说不会设置溢出标志。但是，因为对于无符号数来说，答案小于零，所以设置进位标志。

和其他标志不同，有可以专门修改进位标志的指令。下表介绍这些指令。

| 指　令 | 描　述 |
|---|---|
| CLC | 清空进位标志（设置它为零） |
| CMC | 对进位标志求反（把它改变为相反的值） |
| STC | 设置进位标志（设置它为1） |

这些指令都直接修改EFLAGS寄存器中的进位标志位。

## 6.4 循环

循环是改变程序内指令路径的另外一种方式。循环可以使用单一循环函数编写重复性任务的代码。循环操作重复地执行，直到满足特定条件。

下面几节介绍可以使用的不同循环指令，并且给出在汇编语言程序中使用循环的例子。

### 6.4.1 循环指令

在“使用符号标志”一节介绍的程序例子signtest.s中，通过使用跳转指令和递减寄存器值创建了一个循环。IA-32平台提供了在汇编语言程序中实现循环的更简单的机制：循环指令系列。

循环指令使用ECX寄存器作为计数器并且随着循环指令的执行自动递减它的值。下表介绍循环系列中的指令。

| 指　令 | 描　述 |
|---|---|
| LOOP | 循环直到ECX寄存器为零 |
| LOOPE/LOOPZ | 循环直到ECX寄存器为零，或者没有设置ZF标志 |
| LOOPNE/LOOPNZ | 循环直到ECX寄存器为零，或者设置了ZF标志 |

LOOPE/LOOPZ和LOOPNE/LOOPNZ指令提供了监视零标志的附加功能。

这些指令的格式是：

```
loop address
```

其中address是要跳转到的程序代码位置的标签名称。不幸的是，循环指令只支持8位偏移量，所以只能进行短跳转。

循环开始之前，必须在ECX寄存器中设置执行迭代的次数值。这通常使用下面这样的代码完成：

```
    < code before the loop >
    movl $100, %ecx
loop1:
    < code to loop through >
    loop loop1
    < code after the loop >
```

要注意循环内部的代码。如果ECX寄存器被修改了，就会影响循环的操作。在循环内实现函数调用时要格外谨慎，因为函数可能很容易地在程序员无意识的情况下破坏ECX寄存器的值。

循环的额外好处在于它们递减ECX寄存器的值，而不影响EFLAGS寄存器的标志位。当ECX寄存器值到达零时，零标志不会被设置。

### 6.4.2 循环范例

下面是loop.s程序，它演示LOOP指令如何工作：

```
# loop.s - An example of the loop instruction
.section .data
output:
    .asciz "The value is: %d\n"
.section .text
.globl _start
_start:
    movl $100, %ecx
    movl $0, %eax
loop1:
    addl %ecx, %eax
    loop loop1
    pushl %eax
    pushl $output
    call printf
    add $8, %esp
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

loop.s程序计算ECX寄存器中存储的数字序列，然后在控制台中显示它（要使用printf函数，所以记住用C库和动态连接器进行连接）。LOOP指令用于持续地循环调用ADD函数，直到ECX的值为零。

### 6.4.3 防止LOOP灾难

LOOP指令有个常见的问题，这个问题有时候会困扰汇编语言程序员。如果使用loop.s程序并且把ECX寄存器设置为零会怎么样？尝试这样做并且进行观察。下面是我进行尝试时的输出：

```
$ ./loop
The value is: -2147483648
$
```

显然这不是正确的答案。到底是怎么回事？问题在于LOOP指令的行为方式。当执行LOOP指令时，它首先把ECX中的值递减1，然后检查ECX中的值是否为零。使用这个逻辑，如果在LOOP指令之前ECX的值已经为零，LOOP指令会把它递减1，使它成为-1。因为这个值非零，所以LOOP指令继续执行下去，循环回到定义的标签。循环最终会在寄存器溢出时退出，并且显示错误的值。

为了纠正这个问题，需要检查ECX寄存器包含零值时的特殊条件。幸运的是，Intel提供了专门用于这个目的的指令。如果读者还记得前面的“条件分支”一节，里面讲过如果ECX寄存器为零，就使用JCXZ指令执行条件分支。这正是我们需要的解决这个问题的方式。

betterloop.s程序使用JCXZ指令为应用程序提供某种初步的错误检查：

```
# betterloop.s - An example of the loop and jcxz instructions
.section .data
output:
   .asciz "The value is: %d\n"
.section .text
.globl _start
_start:
   movl $0, %ecx
movl $0, %eax
   jcxz done
loop1:
   addl %ecx, %eax
   loop loop1
done:
   pushl %eax
   pushl $output
   call printf
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

betterloop.s在循环开始前添加了单一指令——JCXZ指令，并且使用单一标签引用指令码的结尾。现在，如果ECX寄存器包含零值，JCXZ指令会捕获它，并且立即转到输出段。运行程序，结果显示这样确实解决了问题：

```
$ ./betterloop
The value is: 0
$
```

## 6.5 模仿高级条件分支

如果使用C、C++、Java或者任何其他高级语言进行程序设计，可能使用很多条件语句，这些条件语句看上去和汇编语言中的条件语句完全不同。读者可以使用本章中学到的汇编语言代码模拟高级语言函数。

学习如何使用汇编语言编写高级函数的最简单的方式是查看汇编器如何完成这样的工作。下面几节通过反汇编C语言函数显示它们是如何使用汇编语言完成的。

### 6.5.1 if语句

高级语言中使用的最常见的条件语句是if语句。下面的程序ifthen.c演示C程序中if语句的常见用法:

```
/* ifthen.c - A sample C if-then program */
#include <stdio.h>

int main()
{
   int a = 100;
   int b = 25;
   if (a > b)
   {
      printf("The higher value is %d\n", a);
   } else
      printf("The higher value is %d\n", b);
return 0;
}
```

因为这个练习的目的是查看代码是如何转换为汇编语言的，所以实际的C程序非常简单——仅仅是简单地比较两个已知值。可以使用GNU编译器的-S参数查看生成的汇编语言代码:

```
$ gcc -S ifthen.c
$ cat ifthen.s
        .file   "ifthen.c"
        .section        .rodata
.LC0:
        .string "The higher value is %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $100, -4(%ebp)
        movl    $25, -8(%ebp)
        movl    -4(%ebp), %eax
        cmpl    -8(%ebp), %eax
        jle     .L2
        movl    -4(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        jmp     .L3
.L2:
        movl    -8(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
.L3:
```

```
        movl    $0, (%esp)
        call    exit
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.2 (Debian)"
$
```

这个简单的C函数生成了很多汇编代码！现在读者会明白为什么我希望保持C代码的简单。现在我们可以逐步地查看这些代码，了解它如何进行操作。

下面是代码的第一段：

```
pushl   %ebp
movl    %esp, %ebp
subl    $24, %esp
andl    $-16, %esp
movl    $0, %eax
subl    %eax, %esp
```

它存储EBP寄存器，以便可以使用它作为指向程序中的本地堆栈的指针。然后手动地操作堆栈指针ESP，为把本地变量压入堆栈留下空间。

代码的下一段创建在if语句中使用的两个变量：

```
movl    $100, -4(%ebp)
movl    $25, -8(%ebp)
```

第一条指令手动地把变量a的值传送到堆栈中的位置（EBP寄存器指向的位置之前4字节）。第二条指令手动地把变量b的值传送到堆栈中的下一个位置（EBP寄存器指向的位置之前8字节）。函数中经常使用的这种技术将在第11章中讨论。既然这两个变量都存储在堆栈中了，该执行if语句了：

```
movl    -4(%ebp), %eax
cmpl    -8(%ebp), %eax
jle     .L2
```

首先，变量a的值被传送到EAX寄存器中，然后把这个值和变量b的值（它仍然在本地堆栈中）进行比较。汇编语言代码没有查看if条件a>b，而是查看相反的条件a<=b。如果语句计算结果为“真”，就跳转到.L2标签，这是if语句的“else”部分：

```
.L2:
    movl    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    $.LC0, (%esp)
    call    printf
```

这是输出变量b的答案的代码，它包含在if语句的else部分中。首先，获取变量b的值并且手动地存放到堆栈中，然后输出文本的位置（位于.LC0标签处）被存放到堆栈中。这两个元素都放在堆栈中之后，调用C函数printf显示答案。然后代码执行到结束指令。

如果JLE指令的结果为假，就是说a不小于或者等于b，则不执行跳转。那么执行if语句的“then”部分：

```
movl    -4(%ebp), %eax
movl    %eax, 4(%esp)
movl    $.LC0, (%esp)
call    printf
jmp     .L3
```

这里，变量a和输出文本被加载到堆栈中。然后调用C函数printf显示答案，并且执行跳转到.L3标签。最后，所有执行路径都归结到C函数exit:

```
.L3:
    movl    $0, (%esp)
    call    exit
    .size   main, .-main
    .section        .note.GNU-stack,"",@progbits
    .ident  "GCC: (GNU) 3.3.2 (Debian)"
```

很容易明白汇编语言代码中包含的if-then逻辑。在汇编语言程序中，可以把相同的逻辑应用到任何需要的if-then状况。

最初看上去，汇编语言中if-then逻辑的这种实现方式是退步的。跳转指令以计算结果为“true”作为跳转到“then”段的条件似乎更加容易。使用相反的条件是有原因的，这在本章后面的6.6节中讲解。

用于实现if语句的汇编语言代码如下:

```
if:
   <condition to evaluate>
   jxx  else        ; jump to the else part if the condition is false
<code to implement the "then" statements>
jmp end             ;jump to the end
else:
   < code to implement the "else" statements>
end:
```

当然，这是个实用价值不大的if语句例子。在实际的产品型程序中，要计算的条件会复杂得多。在这些情况下，计算if语句的条件就变得和if语句代码本身一样至关重要了。

不使用单一条件跳转指令，可以使用几个指令，每条指令计算if条件的一个独立部分。例如，C语言的if语句

```
if (eax < ebx) || (eax == ecx) then
```

生成下面的汇编语言代码:

```
if:
   cmpl %eax, %ebx
   jle else
   cmpl %eax, %ecx
   jne else
then:
   < then logic code>
   jmp end
else:
   < else logic code >
end:
```

这个if语句条件需要两个单独的CMP指令。因为逻辑操作符是OR，所以如果任何一条CMP指令计算结果为真，程序就跳转到else标签。如果逻辑操作符为AND，就需要使用一个中间标签，以便确保两个CMP指令计算都为真。

### 6.5.2 for循环

下一个要研究的语句是for循环。首先，我们看一下下面的C程序例子for.c:

```
/* for.c - A sample C for program */
#include <stdio.h>

int main()
{
   int i = 0;
   int j;
   for (i = 0; i < 1000; i++)
   {
      j = i * 5;
      printf("The answer is %d\n", j);
   }
return 0;
}
```

我再一次使用非常简化的C程序演示如何使用汇编语言代码实现for-next循环。下面是GNU编译器生成的汇编语言代码:

```
$ gcc -S for.c
$ cat for.s
        .file   "for.c"
        .section        .rodata
.LC0:
        .string "The answer is %d\n"
        .text
.globl main
        .type   main, @function
        main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $0, -4(%ebp)
        movl    $0, -4(%ebp)
.L2:
        cmpl    $999, -4(%ebp)
        jle     .L5
        jmp     .L3
.L5:
        movl    -4(%ebp), %edx
        movl    %edx, %eax
        sall    $2, %eax
        addl    %edx, %eax
        movl    %eax, -8(%ebp)
        movl    -8(%ebp), %eax
```

```
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L2
.L3:
        movl    $0, (%esp)
        call    exit
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.2 (Debian)"
$
```

和if语句的代码类似，for语句代码首先对ESP和EBP寄存器进行一些内务处理，手动地将EBP寄存器设置为堆栈的开始，并且为函数中使用的变量留下空间。for语句从.L2标签开始:

```
.L2:
    cmpl    $999, -4(%ebp)
    jle     .L5
    jmp     .L3
```

for语句中的条件是在循环的开始设置的。在这个例子中，条件是确定变量是否小于1 000。如果条件为真，执行就跳转到.L5标签，这里是for循环的代码。当条件为假时，执行跳转到.L3标签，这里是结束代码:

for循环的代码如下:

```
.L5:
    movl    -4(%ebp), %edx
    movl    %edx, %eax
    sall    $2, %eax
    addl    %edx, %eax
    movl    %eax, -8(%ebp)
    movl    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    $.LC0, (%esp)
    call    printf
```

第一个变量位置（C代码中的变量i）被传送给EDX寄存器，然后传送给EAX寄存器。接下来的两个指令是数学操作（将在第8章中详细讲解）。SALL指令执行两次EAX寄存器的左移位。这相当于使EAX寄存器中的数乘4。下一条指令把EDX寄存器值和EAX寄存器值相加。现在EAX寄存器包含的值是其原始值乘5（这很巧妙）。

这个值乘5之后，它被存储在为第二个变量（C代码中的变量j）保留的位置中。最后，这个值和输出文本的位置被存放到堆栈中，并且调用C函数printf。

代码的下一部分回到for语句本身的功能:

```
leal    -4(%ebp), %eax
incl    (%eax)
jmp     .L2
```

我们还没有讨论过LEA指令。它把声明的变量的有效内存地址加载到指定的寄存器中。因此，第一个变量（i）的内存位置被加载到EAX寄存器中。下一条指令使用间接寻址模式把EAX

寄存器指向的值递增1。这实际上使变量i加1。之后，执行跳转回for循环的开始，在这里测试i的值，确定它是否小于1 000，并且再次执行整个循环。

通过这个例子，可以了解汇编语言中实现for循环的框架。伪代码如下：

```
for:
   <condition to evaluate for loop counter value>
   jxx forcode     ; jump to the code of the condition is true
   jmp end         ; jump to the end if the condition is false
forcode:
  < for loop code to execute>
  <increment for loop counter>
   jmp for         ; go back to the start of the For statement
end:
```

while循环代码使用的格式和for循环代码类似。尝试在C程序中创建用于测试的while循环，并且查看生成的汇编代码。它看上去和前面显示的for循环代码类似。

## 6.6 优化分支指令

分支指令严重地影响了应用程序的性能。大多数现代的处理器（包括IA-32系列的处理器）利用指令预取缓存提高性能。在程序运行时，指令预取缓存被填充上顺序的指令。

正如第2章中介绍的，乱序引擎试图尽可能快地执行指令，即使程序中前面的指令还没有执行。但是，分支指令对乱序引擎有严重的破坏作用。下面几节介绍现代奔腾处理如何处理分支，以及可以如何改进汇编语言程序的性能。

### 6.6.1 分支预测

在遇到分支指令时，处理器的乱序引擎必须确定要处理的下一条指令。乱序引擎单元利用称为分支预测前端（branch prediction front end）的独立单元确定是否应该跳转到分支。在分支预测前端试图预测分支的操作中，它利用不同的技术。创建包含条件分支的汇编语言代码时，应该意识到处理器有这个特性。

1. 无条件分支

对于无条件分支，不难确定下一条指令，但是根据跳转距离有多远，下一条指令在指令预取缓存中有可能是不存在的。图6-4演示这种情况。

在确定内存中新的指令位置时，乱序引擎必须首先确定指令在预取缓存中是否存在。如果不存在，那么必须清空整个预取缓存，然后从新的位置重新加载指令。这对应用程序的性能而言是代价很高的。

2. 条件分支

条件分支给处理器提出了更大的挑战。对于每个条件分支，分支预测单元必须确定是否采用分支。通常，当乱序引擎准备执行条件分支时，没有足够的信息用来确定肯定会采用哪个分支方向。

作为替换的做法，分支预测算法试图猜测特定的条件分支将采用哪条路径。这是使用规则和学习的历史实现的。分支预测算法使用3个主要规则：

- 假设会采用向后分支
- 假设不会采用向前分支
- 以前曾经采用过的分支会再次采用

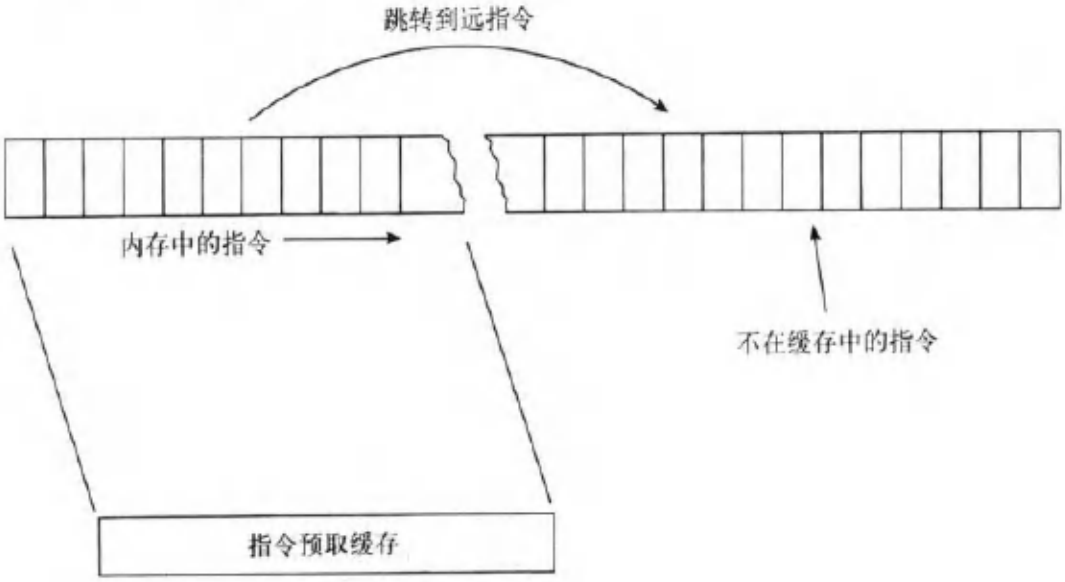



图 6-4

使用一般的程序设计逻辑，最常见的向后分支（跳转到前面的指令码的分支）是在循环中使用的。例如下面的代码片断

```
    movl $100, $ecx
loop1:
    addl %cx, %eax
    decl %ecx
    jns loop1
```

将跳转回loop1标签100次，但是执行下一条指令只有一次。第一条分支原则永远假设将采用向后分支。分支执行的101次里面，只会出现一次方向预测错误。

向前分支有些难于处理。分支预测算法假设大多数情况下条件分支不会采用向前的方向。在程序设计逻辑中，假设紧跟在跳转指令后面的代码最可能被执行，而不是跳转到代码的其他位置。从下面的代码片断可以观察到这一情况：

```
    movl    -4(%ebp), %eax
    cmpl    -8(%ebp), %eax
    jle     .L2
    movl    -4(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    $.LC0, (%esp)
    call    printf
    jmp     .L3
.L2:
    movl    -8(%ebp), %eax
    movl    %eax, 4(%esp)
    movl    $.LC0, (%esp)
    call    printf
.L3:
```

这段代码看上去有些熟悉吗？这是来自对C程序if语句的分析的代码片断。JLE指令后面的代码处理if语句的“then”部分。从分支预测的角度来看，现在我们知道为什么使用JLE指令，而不使用JG指令。当编译器创建汇编语言代码时，它通过猜测if语句的“then”部分比“else”部分更可能被执行，从而试图最优化代码的性能。因为处理器分支预测单元假设不会采用向前跳转，所以“then”代码已经在指令预取缓存中，准备好被执行。

最后一条规则暗示，执行了多次的分支在多数情况下可能采用相同的路径。分支目标缓冲区（Branch Target Buffer，BTB）跟踪处理器执行的每个分支指令，分支的结果存储在缓冲区区域中。

BTB信息高于分支的前两个规则。例如，如果第一次遇到分支时，没有采用向后的方向，分支预测单元就会假设任何后续分支都不会采用向后方向，而不是假设会应用向后分支的规则。

BTB的问题在于它可能被充满。当BTB被充满时，查找分支结果会花费更长时间，并且降低执行分支的性能。

### 6.6.2 优化技巧

处理器尽最大的努力优化处理分支的方式，也可以在汇编语言程序中使用几个技巧来帮助处理器。下面几节介绍一些分支优化的技巧，这些技巧是Intel推荐在奔腾系列的处理器上使用的。

1. 消除分支

解决分支性能问题的最显而易见的方式是尽可能地消除分支的使用。Intel提供一些专门的指令帮助达到这个目的。

在第5章“传送数据”中讨论过CMOV指令。这些指令被专门设计为帮助汇编语言程序员避免使用分支设置数据值。下面是使用CMOV指令的一个例子：

```
movl value, %ecx
cmpl %ebx, %ecx
cmova %ecx, %ebx
```

CMOVA指令检查CMP指令的结果。如果ECX寄存器中的无符号整数值大于EBX寄存器中的无符号整数值，就把ECX寄存器中的值存放到EBX寄存器中。这一功能使我们可以创建cmovtest.s程序，它确定一系列数字中的最大者，而且不使用大量跳转指令。

有时候重复几个额外指令能够消除跳转。这种小的指令开销将容易地适合指令预取缓存，并且补偿跳转本身造成的性能影响。这种方式的典型例子是循环内出现分支的情况：

```
loop:
   cmp data(, %edi, 4), %eax
   je part2
   call function1
   jmp looptest
part2:
   call function2
looptest:
   inc %edi
   cmpl $10, %edi
   jnz loop
```

根据从data数组读出的值，循环调用两个函数之一。调用函数之后，跳转到循环的末尾，递增数组的下标值并且返回循环的开始。每次调用第一个函数时，必须计算JMP指令以确定是否向

前跳转到looptest标签。因为这是向前分支，所以预测不会采用它，这就会导致性能损失。

为了改变这种情况，可以把代码片断修改如下：

```
loop:
   cmp data(, %edi, 4), %eax
   je part2
   call function1
   inc %edi
   cmp $10, %edi
   jnz loop
   jmp end
part2:
   call function2
   inc %edi
   cmp $10, %edi
   jnz loop
end:
```

在循环内不使用向前分支，looptest的代码被复制到第一个函数代码段之内，这就从代码中消除了一个向前跳转。

2. 首先编写可预测分支的代码

可以利用分支预测单元的规则提高应用程序的性能。就像在前面if语句中看到的，把最可能采用的代码安排在向前跳转的顺序执行语句中，会提高需要它时它在指令预取缓存中的可能性。允许跳转指令跳转到使用的可能性低一些的代码段。

对于使用向后分支的代码，要试图使用向后分支路径作为最可能被采用的路径。实现循环时这通常不是问题，但是在某些情况下也许必须改变程序逻辑以便实现这个目的。

图6-5总结这些情况。

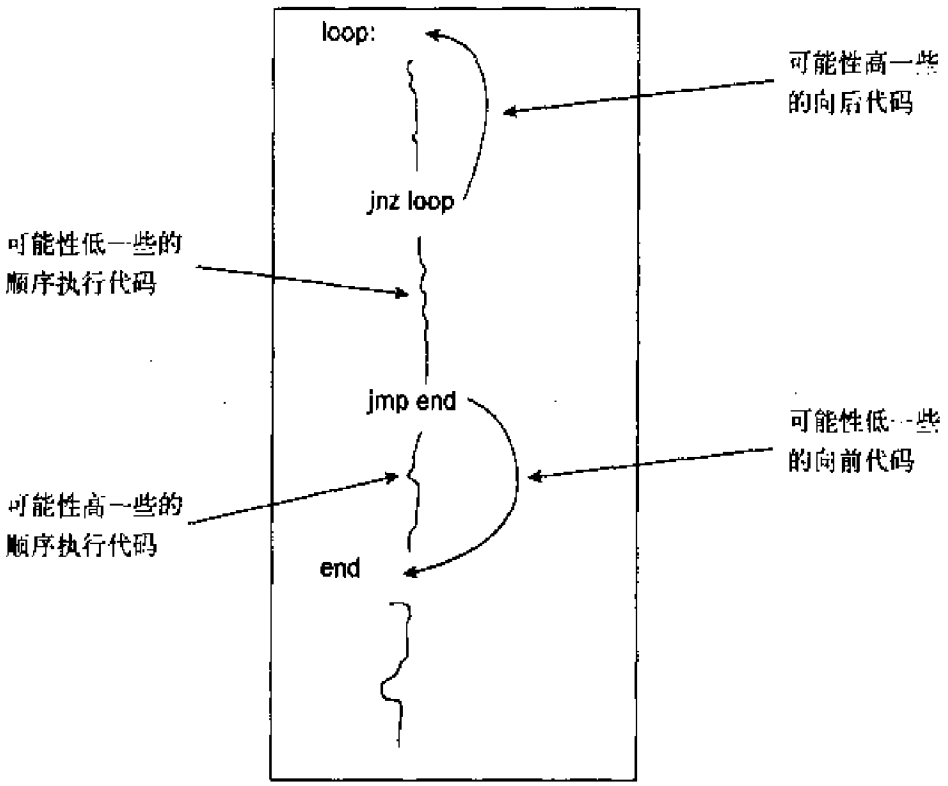



图 6-5

3. 展开循环

虽然循环一般都可以通过向后分支规则预测，但是，即使正确地预测了分支，仍然有性能损失。更好的经验规则是尽可能地消除小型循环。

问题出现在循环的开销上。即使是简单的循环也需要每次迭代时都必须检查的计数器，还有必须计算的跳转指令。根据循环内的程序逻辑指令的数量，这可能是很大的开销。

对于比较小的循环，展开循环能够解决这个问题。展开循环意味着手动地多次编写每条指令的代码，而不是使用循环返回相同的指令。下面的代码是可以展开的小循环的例子：

```
   movl values, %ebx
   movl $1, %edi
loop:
   movl values(, %edi, 4), %eax
   cmp %ebx, %eax
   cmova %eax, %ebx
   inc %edi
   cmp $4, %edi
   jne loop
```

这是第5章中cmovtest.s程序中的主循环。不通过4次循环执行指令来查找最大值，你可以把循环展开为4个传送：

```
movl values, %ebx
movl $values, %ecx
movl (%ecx), %eax
cmp %ebx, %eax
cmova %eax, %ebx
movl 4(%ecx), %eax
cmp %ebx, %eax
cmova %eax, %ebx
movl 8(%ecx), %eax
cmp %ebx, %eax
cmova %eax, %ebx
movl 12(%ecx), %eax
cmp %ebx, %eax
cmova %eax, %ebx
```

虽然指令的数量有很大增加，但是处理器能够把所有这些指令都存放到指令预取缓存中，并且顺畅迅速地执行它们。

*在展开循环时要小心，因为可能展开过多的指令并且过度地填充预取缓存。这将迫使处理器不断地填充和清空预取缓存。*

## 6.7 小结

本章介绍的指令帮助读者在汇编语言程序中进行程序逻辑设计。几乎每个汇编语言程序都需要这样的能力：根据数据值跳转到指令码的其他部分，或者按照特定次数循环执行代码段。

IA-32平台提供几个用于编写分支和循环函数的指令。分支函数有两个不同类型：无条件分支和条件分支。无条件分支的执行不受外部值或者事件的影响，而条件分支依赖外部值或者事件来决定是否跳转。

无条件分支有3种类型：跳转、调用和中断。无条件跳转是执行控制的最基本形式。JMP指

令强制指令指针改变到目的位置，并且处理器执行这个位置上的下一条指令。调用和跳转类似，但是它支持在调用之后返回原始位置的能力。返回位置被存储在堆栈区域中，被调用的函数在返回发出调用的区域之前，必须把堆栈恢复到它的原始状态。软件中断用于提供访问操作系统中的低级内核函数的能力。Microsoft Windows和Linux都通过软件中断提供系统调用。Linux系统调用通过软件中断0x80使用。

条件分支依赖EFLAGS寄存器的值。进位、溢出、奇偶校验、符号和零标志专门用于影响条件分支。特定的分支指令监视特定的标志位，比如当进位标志被设置为1时，JC指令就进行跳转；当零标志被设置为1时，JZ指令就进行跳转。为条件跳转指令比较两个值并且设置EFLAGS位时，可以使用CMP指令。

循环提供了轻松地重复代码功能，而且不必复制很多代码的方法。就像高级语言中一样，通过每次迭代时递减计数器的值，循环可以按照特定次数执行任务。LOOP指令自动地使用ECX寄存器作为计数器，在每次迭代的过程中递减并且测试这个计数器。

可以使用一般的汇编语言跳转和循环来模仿高级语言的条件功能，比如if-then和for语句。为了了解C函数是如何编码的，可以使用GNU编译器的-S参数查看生成的汇编语言代码。

使用奔腾处理器时，可以使用一些优化技术来提高汇编语言代码的性能。奔腾处理器使用指令预取缓存，并且试图尽可能快地把指令加载到缓存中。不幸的是，分支指令可能对预取缓存造成有害影响。在缓存中检测到分支时，乱序引擎会试图预测分支最可能采用的路径。如果预测错误，指令预取缓存就会加载用不到的指令，并且浪费处理器的时间。为了帮助解决这一问题，读者应该了解处理器是怎样预测分支的，并且尝试按照相同的方式编写分支的代码。另外，尽可能地消除分支将显著地加快执行速度。最后，消除循环并且把它们转换为一系列顺序的操作，使处理器能够把所有指令加载到预取缓存中，并且不必担心用于循环的分支。

下一章讨论处理器如何处理各种类型的数字。就像高级语言一样，在汇编语言程序中有多种表示数字的方式，并且有多种处理这些数字的方式。在处理数学密集的操作时，了解数字格式的所有不同类型将有所帮助。

# 第7章 使用数字

表示和处理数字是汇编语言程序工作的很大一部分。几乎每个应用程序都使用某种类型的数字数据来处理信息。就像高级语言一样，汇编语言也以很多不同的格式表示数字。如果使用C或者C++进行过程序设计，读者就会熟悉定义特定数据类型的变量。每次出现变量时，编译器都知道它表示的是什么数据类型。在汇编语言程序设计中，情况并不总是如此。存储在内存或者寄存器中的值可以被解释为很多不同的数据类型。作为汇编语言程序员，任务是确保使用正确的指令、以正确的方式解释存储的数据。本章的目的是介绍汇编语言程序中可用的不同数字格式，并且演示如何使用它们。

本章的开始部分介绍整数数据类型，包括无符号和带符号。之后，讨论特殊的二进制编码的十进制（Binary Coded Decimal）数据类型，并且提供在程序中使用它们的范例。之后，介绍浮点数，包括标准的单精度和双精度浮点格式，还有Intel双精度扩展格式以及打包的单精度和双精度格式。最后，介绍一些把数字数据从一种格式转换为另一种格式的Intel指令。

## 7.1 数字数据类型

在汇编语言程序中，有众多表示数字值的方式。通常在汇编语言程序中，必须使用多种数据类型来表示数据元素。IA-32平台包含可以在汇编语言程序中使用的几种不同的数字数据类型。核心的数字数据类型如下：

- 无符号整数
- 带符号整数
- 二进制编码的十进制
- 打包的二进制编码的十进制
- 单精度浮点数
- 双精度浮点数
- 双精度扩展浮点数

除了基本的数字数据类型之外，奔腾处理器的SIMD扩展还添加了其他高级数字数据类型：

- 64位打包整数
- 128位打包整数
- 128位打包单精度浮点数
- 128位打包双精度浮点数

虽然可用的数字数据类型的清单很长，但是在汇编语言中处理不同数据类型的数字是相对容易的。下面几节介绍每种数据类型，并且给出如何在汇编语言程序中使用它们的范例。

## 7.2 整数

汇编语言程序中使用的最基本的数字形式是整数。它们可以表示一个很大范围内的全部值。本节介绍汇编语言程序中可以使用的不同整数类型，并且介绍处理器如何处理各种不同类型的整数值。

### 7.2.1 标准整数长度

可以用各种各样的长度表示整数——就是说，用于表示整数数量的字节数目是各种各样的。基本的IA-32平台支持4种不同的整数长度：

- 字节（Byte）：8位
- 字（Word）：16位
- 双字（Doubleword）：32位
- 四字（Quadword）：64位

要记住，存储在内存中的超过1个字节的整数被存储为小尾数（little-endian）格式，这很重要。这就是说，低位字节存储在最低的内存位置，其余字节顺序地存储在它之后。但是，把整数值传送给寄存器时，值按照大尾数（big-endian）格式存储在寄存器中（见图7-1）。有时候这会给处理造成混乱。

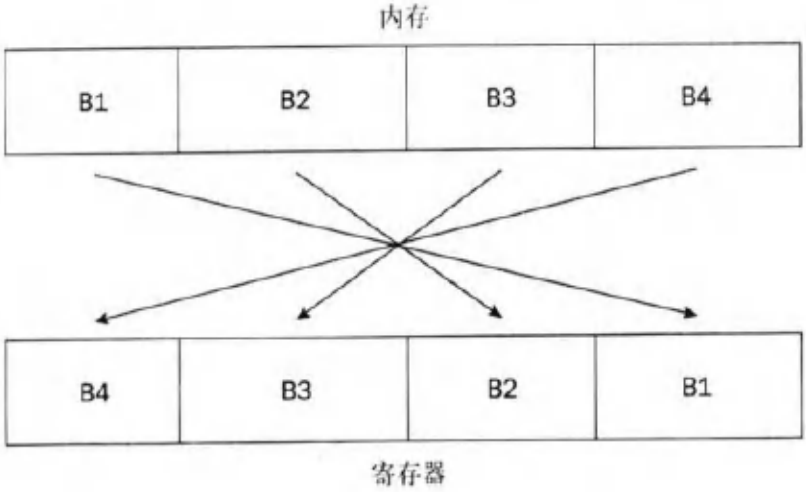



图 7-1

这一转换在处理器中悄悄进行，所以不必担心转换问题，但是当调试应用程序并且查看数据值时，这很重要。可能见到下面这样的情况：

```
(gdb) x/x &data
0x80490bc <data>:       0x00000225
(gdb) x/4b &data
0x80490bc <data>:       0x25    0x02    0x00    0x00
(gdb) print/x $eax
$1 = 0x225
(gdb)
```

十进制值549存储在内存位置data中，它被传送给EAX寄存器。第一个gdb命令使用x命令以

十六进制格式显示位于data标签的内存位置中的值。十六进制显示我们期望的549的十六进制版本。下一条命令显示构成这个整数值的4个字节。注意二进制格式版本以相反的顺序显示十六进制值0x25和0x02，这是我们期望的小尾数格式。最后一条命令使用print命令再次以十六进制格式显示被加载到EAX寄存器中的相同的值。

### 7.2.2 无符号整数

无符号整数几乎是“所见即所得”的数据类型。组成整数的字节的值直接表示整数值。

根据使用的位的数目，4种不同长度的无符号整数可以生成4种不同数值范围的无符号整数。下表列出了这些长度。

| 位 | 整数值 |
|---|---|
| 8 | 0到255 |
| 16 | 0到65 535 |
| 32 | 0到4 294 967 295 |
| 64 | 0到18 446 744 073 709 551 615 |

8位整数值包含在单一字节之内（正如预期）。字节中包含的二进制值就是实际的整数值。因此，二进制值为11101010的字节（可以表示为十六进制值0xEA）的无符号整数值是234。

16位无符号整数值包含在两个连续的字节中，它们组合在一起构成一个字（word）。图7-2显示存储在寄存器中的字值的例子。

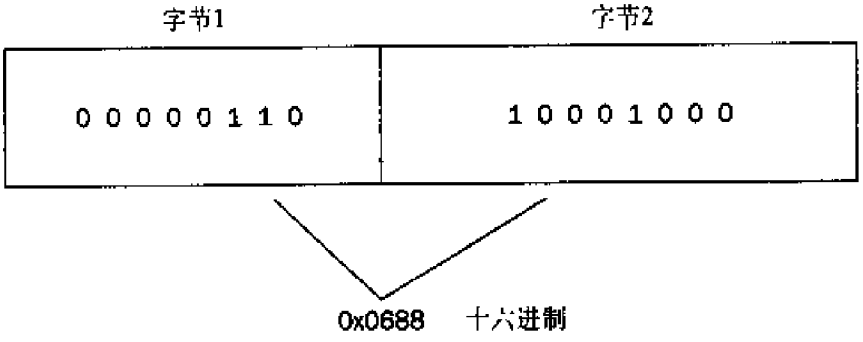



图 7-2

32位无符号整数值（以小尾数格式）包含在4个连续的字节中，它们组合在一起构成双字。双字是IA-32平台上最常用的无符号整数格式。图7-3显示的是双字的例子。

在图7-3中，每个字节由一对十六进制数字表示（每个十六进制值是4位），它们组合在一起构成8个字符的十六进制值。同样，这个例子使用大尾数格式，就像在寄存器中那样。

64位无符号整数值包含在8个连续的字节中，它们组合在一起构成四字。图7-4显示的是四字的例子。

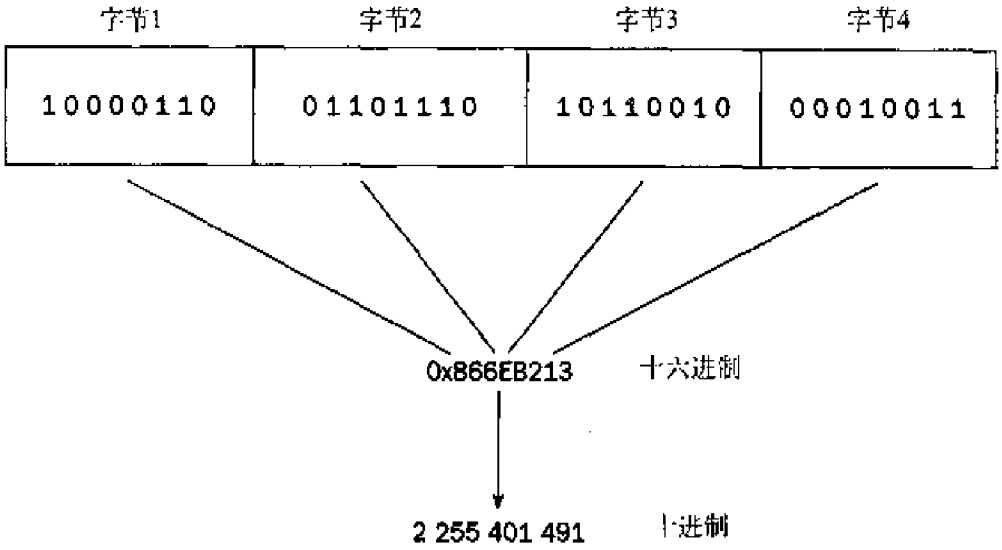



图 7-3

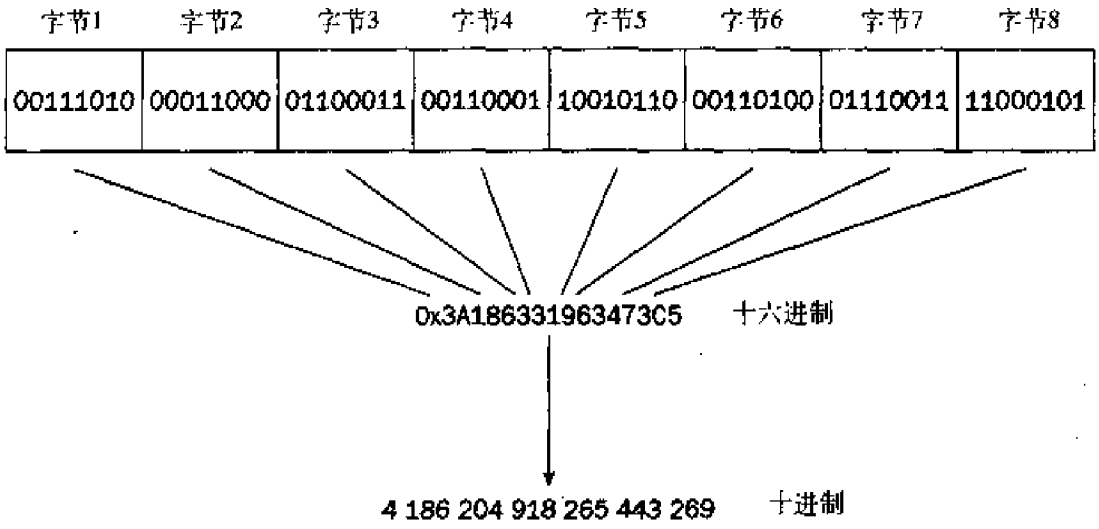



图 7-4

### 7.2.3 带符号整数

虽然使用无符号整数很容易，但是其缺陷是没有办法表示负数。为了解决这个问题，在处理器上需要采用能够表示负数的方法。有3种方法用于在计算机中描述负数：

- 带符号数值
- 反码（One's complement）
- 补码（Two's complement）

所有这3种方法都使用和无符号整数相同的位长度（字节、字、双字和四字），但是在位中表示十进制值的方式是不同的。IA-32平台使用补码方式表示带符号整数，但是了解每种方法如何工作是有好处的。下面几节介绍每种方法。

1. 带符号数值

带符号数值的方法把组成带符号整数的位分为两部分：符号位和数值位。字节的最大有效位

（最左侧的一位）用于表示值的符号。正数的最大有效位包含0，而负数的这个位置是1。值中的其余位使用一般的二进制值表示数字的数值，如图7-5所示。

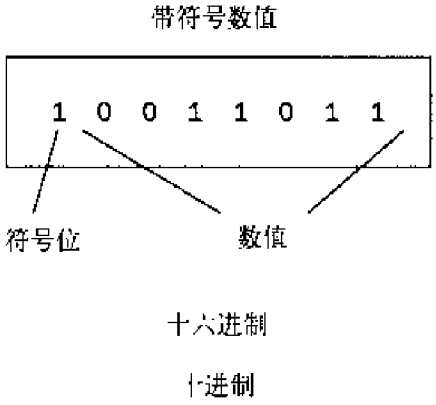



图 7-5

带符号数值的一个问题是有两种不同的表示0值的方式：00000000（十进制的+0）和10000000（十进制的−0）。这使一些数学处理变得复杂。另外，使用带符号数值的数字的数学运算是复杂的，因为简单的带符号整数的加法和减法不能按照无符号数字的方式进行。例如，简单的加法00000001（十进制1）和10000001（十进制−1）相加得到10000010（十进制−2），这不是正确的答案。这要求处理器为带符号整数和无符号整数提供不同的数学运算指令。

2. 反码

反码方法采用无符号整数的相反代码生成相应的负值。求反把所有为0的位改变为1，把所有为1的位改变为0。因此，00000001的反码就是11111110。同样，对于带符号数字，当执行数学操作时，反码数字会产生一些问题。有两种方式可以表示零值（00000000和11111111），反码数字的数学运算也是复杂的（它不允许进行标准二进制运算）。

3. 补码

补码通过使用简单的数学技巧，解决了带符号数值和反码方法的数学运算问题。对于负整数值，值的反码加上1就是它的补码。

例如，为了得到十进制−1的补码，可以这样做：

1) 得到00000001的反码，结果是11111110。

2) 反码加上1，结果是11111111。

对−2值进行相同处理，会得到11111110，−3会得到11111101。读者也许注意到这里的规律。补码值从11111111（十进制的−1）开始递减，直到到达10000000，它表示−128。当然，对于多字节整数长度，相同的规则跨字节适用。

虽然这样做看上去有些奇怪，但是它解决了带符号整数的加法和减法的所有问题。例如，值00000001（+1）和11111111（−1）相加得到00000000，并且带有进位值。在整数运算中进位值被忽略，所以最终的值确实是0。相同的硬件可以同时用于无符号值和带符号值的加法和减法操作。

对于相同的位数，补码格式表示的值的数量和无符号整数对应的值的数量是相同的，但是必须把值分为正值和负值。因此，带符号整数的最大值是无符号值的一半。下表列出每种带符号整数的最小值和最大值。

| 位 | 最小和最大带符号值 |
|---|---|
| 8 | −128到127 |
| 16 | −32 768到32 767 |
| 32 | −2 147 483 648到2 147 483 647 |
| 64 | −9 223 372 036 854 775 808到9 223 372 036 854 775 807 |

### 7.2.4 使用带符号整数

内存或者寄存器中表示的带符号整数经常是难以识别的，除非知道期望的是什么。有时候GNU调试器能够有所帮助，但是有时候甚至它也会混淆。程序范例inttest.s演示这种情况：

```
# inttest.s - An example of using signed integers
.section .data
data:
   .int -45
.section .text
.globl _start
_start:
   nop
   movl $-345, %ecx
   movw $0xffb1, %dx
   movl data, %ebx
   movl $1, %eax
   int $0x80
```

inttest.s程序演示把带符号整数存储在寄存器中的3种不同方式。前2种使用立即值把带符号负整数存放在寄存器中：

```
movl $-345, %ecx
movw $0xffb1, %dx
```

MOVW指令把16位的带符号整数值0xFFB1（-79）存放在DX寄存器中。第三种方法使用data标签（它包含一个带符号整数值），并且把它存放到EBX寄存器中。

对程序进行汇编之后，可以在调试器中运行它，查看执行情况。单步运行指令，直到所有数据都被加载到寄存器中。下一步，使用info命令显示寄存器值：

```
(gdb) info reg
eax            0x0      0
ecx            0xfffffea7       -345
edx            0xffb1   65457
ebx            0xffffffd3       -45
```

调试器假设EBX和ECX寄存器包含带符号整数，并且使用我们期望的数据类型显示答案。不幸的是，EDX寄存器出现了问题。因为调试器试图把整个EDX寄存器作为带符号整数数据值显示，所以它假设整个EDX寄存器包含一个双字带符号整数（32位）。因为EDX寄存器只包含一个单字整数（16位），所以解释出的值是错误的。记住，寄存器中的数据仍然是正确的（0xFFB1），但是调试器认为的这个数字表示的内容是错误的。

### 7.2.5 扩展整数

inttest.s程序中演示的左右为难的情况表现出处理器在同时使用不同整数长度的情况下如何处理带符号整数。读者常常会发现使用一种长度的整数值，并且需要把这个值传送到长度大一些的位置（比如把字传送给双字）。虽然这看上去是微不足道的小事情，但是有时候并不是这样简单。

1. 扩展无符号整数

把无符号整数值转换为位数更大的值时（比如把字转换为双字），必须确保所有的高位部分

都被设置为零。不应该简单地把一个值复制给另一个值，比如：

```
movw %ax, %bx
```

这样不能保证EBX寄存器的高位部分包含零。为了完成这个操作，必须使用两条指令：

```
movl $0, %ebx
movw %ax, %ebx
```

MOVL指令用于把零值加载到EBX寄存器中。这样保证EBX寄存器的每一位都是0。然后可以安全地把AX寄存器中的无符号整数值传送给EBX寄存器。

为了帮助程序员应付这种情况，Intel提供了MOVZX指令。这条指令把长度小的无符号整数值（可以在寄存器中，也可以在内存中）传送给长度大的无符号整数值（只能在寄存器中）。

MOVZX指令的格式如下：

```
movzx source, destination
```

其中source可以是8位或16位寄存器或者内存位置，destination可以是16位或者32位寄存器。movzxtest.s程序演示这条指令的使用：

```
# movzxtest.s - An example of the MOVZX instruction
.section .text
.globl _start
_start:
   nop
   movl $279, %ecx
   movzx %cl, %ebx
   movl $1, %eax
   int $0x80
```

movzxtest.s程序简单地把一个大的值存放到ECX寄存器中，然后使用MOVZX指令把低8位复制到EBX寄存器。因为存放在ECX寄存器中的值使用长度为字的无符号整数表示它（它大于255），所以CL中的值只表示完整值的一部分。可以在调试器中监视程序的运行并且查看寄存器的值是如何变化的：

```
$ gdb -q movzxtest
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file movzxtest.s, line 5.
(gdb) run
Starting program: /home/rich/palp/chap07/movzxtest

Breakpoint 1, _start () at movzxtest.s:5
5          movl $279, %ecx
Current language:  auto; currently asm
(gdb) s
6          movzx %cl, %ebx
(gdb) s
7          movl $1, %eax
(gdb) print $ecx
$1 = 279
(gdb) print $ebx
$2 = 23
(gdb) print/x $ecx
$3 = 0x117
(gdb) print/x $ebx
$4 = 0x17
(gdb)
```

通过输出EBX和ECX寄存器的十进制值，马上就能发现无符号整数值没有被正确地复制——原始值为279，但是新的值只是23。通过按照十六进制显示值，可以发现为什么会这样。十六进制格式的原始值为0x0117，它占用一个双字。MOVZX指令只传送了ECX寄存器的低位字节，而用0填充了EBX中剩余的字节，这样就在EBX寄存器中生成了0x17这个值。

2. 扩展带符号整数

扩展带符号整数值和扩展无符号整数是不同的。使用零填充高位会改变负数的数据值。例如，把值-1（11111111）传送给双字会生成值0000000011111111，在带符号整数表示法中它是+127，而不是-1。为了使带符号扩展能够起作用，新添加的位必须被设置为1。因此，新的双字将生成值1111111111111111，这是带符号整数表示法的值-1，这是正确的。

Intel提供了MOVSX指令，它允许扩展带符号整数并且保留符号。它和MOVZX指令类似，但是它假设要传送的字节是带符号整数格式，并且试图在传送过程中保持带符号整数的值不变。movsxtest.s程序演示这种情况：

```
# movsxtest.s - An example of the MOVSX instruction
.section .text
.globl _start
_start:
   nop
   movw $-79, %cx
   movl $0, %ebx
   movw %cx, %bx
   movsx %cx, %eax
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

movsxtest.s程序在CX寄存器中（双字长度）定义一个负值。然后试图把这个值复制到EBX寄存器中，程序首先使用零填充EBX寄存器，然后使用MOV指令。下一步，使用MOVSX指令把CX寄存器的值传送给EAX寄存器。为了查看执行情况，必须在调试器中运行程序，并且显示寄存器值：

```
(gdb) info reg
eax            0xffffffb1       -79
ecx            0xffb1   65457
edx            0x0      0
ebx            0xffb1   65457
```

单步运行程序，一直运行到MOVSX指令之后，可以使用调试器的info命令显示寄存器值。ECX寄存器包含的值是0x0000FFB1。低16位包含的值是0xFFB1，它是带符号整数格式的-79。当CX寄存器被传送给EBX寄存器时，EBX寄存器包含的值是0x0000FFB1，它是带符号整数格式的65 457，这不是我们希望的。

使用MOVSX指令把CX寄存器传送给EAX寄存器之后，EAX寄存器包含的值是0xFFFFFFB1，它是带符号整数格式的-79。MOVSX指令正确地为这个值添加了高位部分的1。

为了确保我们的做法是正确的，movsxtest2.s程序完成相同的工作，但是这一次使用带符号整数正值：

```
# movsxtest2.s - Another example using the MOVSX instruction
```

```
.section .text
.globl _start
_start:
   nop
   movw $79, %cx
   xor %ebx, %ebx
   movw %cx, %bx
   movsx %cx, %eax
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

汇编和连接程序之后，在调试器中运行它并且查看寄存器的值：

```
(gdb) info reg
eax            0x4f     79
ecx            0x4f     79
edx            0x0      0
ebx            0x4f     79
```

这一次，当CX寄存器被传送给空的EBX寄存器时，值的格式是正确的（因为高位部分的零对于正数是没有问题的）。另外，MOVSX指令正确地使用零填充了EAX寄存器，生成了正确的32位带符号整数值。

### 7.2.6 在GNU汇编器中定义整数

前面小节中的范例程序演示如何在汇编语言程序中使用立即数值。也可以在数据段中使用命令定义带符号整数值。

第5章介绍过如何在数据段中使用.int、.short和.long命令定义带符号整数值。这些命令创建双字的带符号整数值。还可以使用.quad命令创建四字的带符号整数值。

.quad命令可以定义一个或者多个带符号整数值，但是为每个值分配8个字节。为了演示这一情况，使用quadtest.s程序：

```
# quadtest.s - An example of quad integers
.section .data
data1:
   .int 1, -1, 463345, -333252322, 0
data2:
   .quad 1, -1, 463345, -333252322, 0
.section .text
.globl _start
_start:
   nop
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

quadtest.s程序简单地在标签data1的位置定义一个包含5个双字带符号整数的数组，在标签data2的位置定义一个包含5个四字带符号整数的数组（使用.quad命令），然后退出程序。为了查看执行情况，再次对程序进行汇编并且在调试器中运行它。

首先，查看调试器认为data1和data2数组的十进制值是什么：

```
(gdb) x/5d &data1
0x8049084 <data1>:      1       -1      463345  -333252322
0x8049094 <data1+16>:   0
(gdb) x/5d &data2
0x8049098 <data2>:      1       0       -1      -1
0x80490a8 <data2+16>:   463345
(gdb)
```

data1数组的值正如我们期望的，但是查看data2数组中的值怎么样呢？这不是程序中使用的值。问题在于调试器假设这些值是双字的带符号整数值。

下一步，查看内存中标签data1位置的数组值是如何存储的：

```
(gdb) x/20b &data1
0x8049084 <data1>:    0x01    0x00    0x00    0x00    0xff    0xff    0xff    0xff
0x804908c <data1+8>:  0xf1    0x11    0x07    0x00    0x1e    0xf9    0x22    0xec
0x8049094 <data1+16>: 0x00    0x00    0x00    0x00
(gdb)
```

这正是我们期望的——每个数组元素使用4个字节，并且按照小尾数格式存放。现在，查看存储在标签data2位置的数组值：

```
(gdb) x/40b &data2
0x8049098 <data2>:    0x01    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x80490a0 <data2+8>:  0xff    0xff    0xff    0xff    0xff    0xff    0xff    0xff
0x80490a8 <data2+16>: 0xf1    0x11    0x07    0x00    0x00    0x00    0x00    0x00
0x80490b0 <data2+24>: 0x1e    0xf9    0x22    0xec    0xff    0xff    0xff    0xff
0x80490b8 <data2+32>: 0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
(gdb)
```

我们通知过汇编器，标签data2位置的数据值是使用四字编码的，所以每个值使用8个字节。的确，汇编器把这些值存放到了正确的位置，但是调试器不知道仅仅通过x/d命令如何显示这些值。

如果希望在调试器中显示四字的带符号整数值，就必须使用gd选项：

```
(gdb) x/5gd &data2
0x8049098 <data2>:      1       -1
0x80490a8 <data2+16>:   463345  -333252322
0x80490b8 <data2+32>:   0
(gdb)
```

这次，情况正常了。

## 7.3 SIMD整数

Intel的单指令多数据（Single Instruction Multiple Data，SIMD）技术提供了定义整数的其他方式（参见第2章“IA-32平台”）。这些新的整数类型使处理器可以同时对一组多个整数执行数学运算操作。

SIMD架构使用打包的整数数据类型。打包的整数是能够表示多个整数值的一系列字节。可以把字节系列看作一个整体，对它执行数学操作，并行地处理系列中的各个整数值（这一概念在第17章“使用高级IA-32特性”中讲解）。下面几节讲解奔腾处理器上可用的不同SIMD打包整数类型。

### 7.3.1 MMX整数

多媒体扩展（Multimedia Extension，MMX）技术是在奔腾MMX和奔腾II处理器中引入的，它提供3种新的整数类型：

- 64位打包字节整数
- 64位打包字整数
- 64位打包双字整数

以上每种数据类型都提供把多个整数数据元素包含到（或者说打包到）单一的64位MMX寄存器中的能力。图7-6演示如何把每种数据类型填充到64位寄存器中。

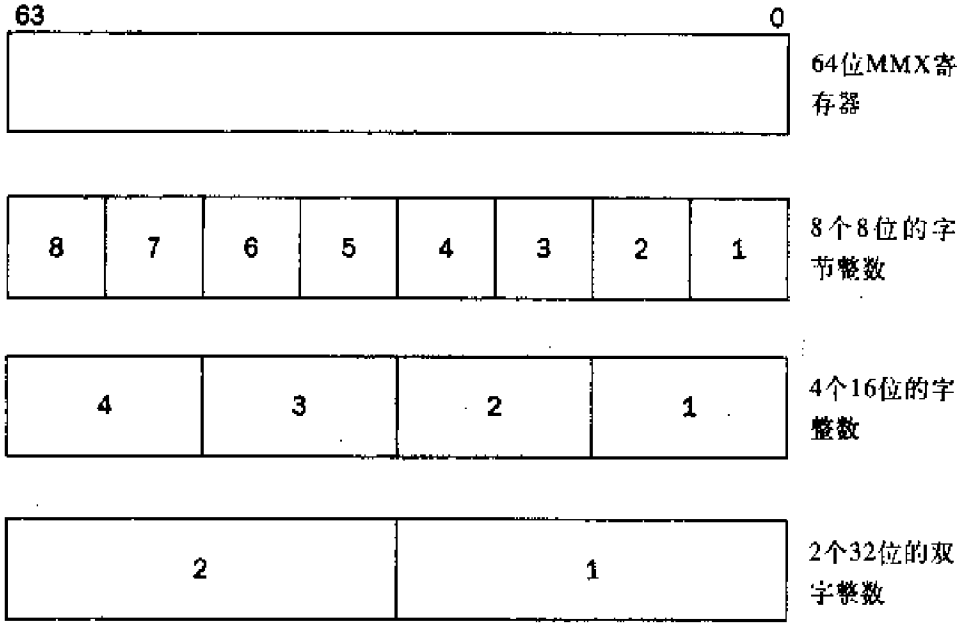



图 7-6

如图7-6所示，8个字节整数、4个字整数或者2个双字整数都可以打包到单一的64位MMX寄存器中。

正如第2章中讨论过的，MMX寄存器被映射到FPU寄存器，所以使用MMX寄存器时要小心。记住，在使用任何MMX寄存器指令之前，都要把FPU寄存器中存储的所有数据保存到内存中。这在本章后面的7.5.5节中讲解。

MMX平台提供了对MMX寄存器中打包的每个整数值执行并行数学操作的附加指令。

### 7.3.2 传送MMX整数

可以使用MOVQ指令把数据传送到MMX寄存器中，但是必须决定当前应用程序将使用3种打包整数格式中的哪一种。MOVQ指令的格式如下：

```
movq source, destination
```

其中source和destination可以是MMX寄存器、SSE寄存器或者64位的内存位置（但是不能在内存位置之间传送MMX整数）。

mmxtest.s程序演示把双字和字节整数加载到MMX寄存器中：

```
# mmxtest.s - An example of using the MMX data types
.section .data
values1:
   .int 1, -1
values2:
   .byte 0x10, 0x05, 0xff, 0x32, 0x47, 0xe4, 0x00, 0x01
.section .text
.globl _start
_start:
   nop
   movq values1, %mm0
   movq values2, %mm1
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

mmxtest.s程序定义两个数据数组。第一个数组（values1）定义2个双字带符号整数，第二个数组（values2）定义8个字节带符号整数值。使用MOVQ指令把这些值加载到前2个MMX寄存器中。

对源代码进行汇编之后，可以在调试器中监视程序的运行。单步运行到MOVQ指令之后，可以显示MM0和MM1寄存器中的值：

```
(gdb) print $mm0
$1 = {uint64 = -4294967295, v2_int32 = {1, -1}, v4_int16 = {1, 0, -1, -1},
  v8_int8 = "\001\000\000\000ÿÿÿÿ"}
(gdb) print $mm1
$2 = {uint64 = 72308588487312656, v2_int32 = {855573776, 16835655},
  v4_int16 = {1296, 13055, -7097, 256}, v8_int8 = "\020\005ÿ2Gä\000\001"}
(gdb) print/x $mm1
$3 = {uint64 = 0x100e44732ff0510, v2_int32 = {0x32ff0510, 0x100e447},
  v4_int16 = {0x510, 0x32ff, 0xe447, 0x100}, v8_int8 = {0x10, 0x5, 0xff, 0x32,
    0x47, 0xe4, 0x0, 0x1}}
(gdb)
```

在奔腾处理器上，MMX寄存器被映射到现有的FPU寄存器，所以根据所使用的gdb的版本，在调试器中显示寄存器的信息可能有些困难。在gdb的老版本中，不能够直接显示MMX寄存器，必须显示它们对应的FPU寄存器。mm0寄存器被映射到第一个FPU寄存器——st0r，mm1寄存器被映射到第二个FPU寄存器——st1（这在第9章中讲解）。不幸的是，调试器不知道如何解释FPU寄存器中的数据，所以必须把数组显示为原始的十六进制值并且自己解释它们。

如果使用新版本的GNU调试器，就可以直接显示MMX寄存器，如前面的代码所示。显示寄存器时，调试器不知道寄存器中数据的格式是什么，所以它会显示所有可能的情况。第一个print命令把MM0寄存器的内容显示为双字整数值。因为前面的例子使用双字整数值，所以唯一有意义的显示格式是int32，它显示正确的信息。可以使用print/f命令使调试器只生成这一格式。

不幸的是，因为MM1寄存器包含字节整数值，所以不能按照十进制模式显示它。替换的方法是，可以使用print命令的x参数显示寄存器中的原始字节。使用这条命令，可以看到各个字节被正确地存放到了MM1寄存器中。

### 7.3.3 SSE整数

流化SIMD扩展（Streaming SIMD Extension，SSE）技术（也是在第2章中介绍过的）提供

用于处理打包数据的8个128位XMM寄存器（名为XMM0到XMM7）。SSE2技术（奔腾4处理器中引入的）提供4种额外的打包带符号整数数据类型：

- 128位打包字节整数
- 128位打包字整数
- 128位打包双字整数
- 128位打包四字整数

这些值被打包在128位XMM寄存器中，如图7-7所示：

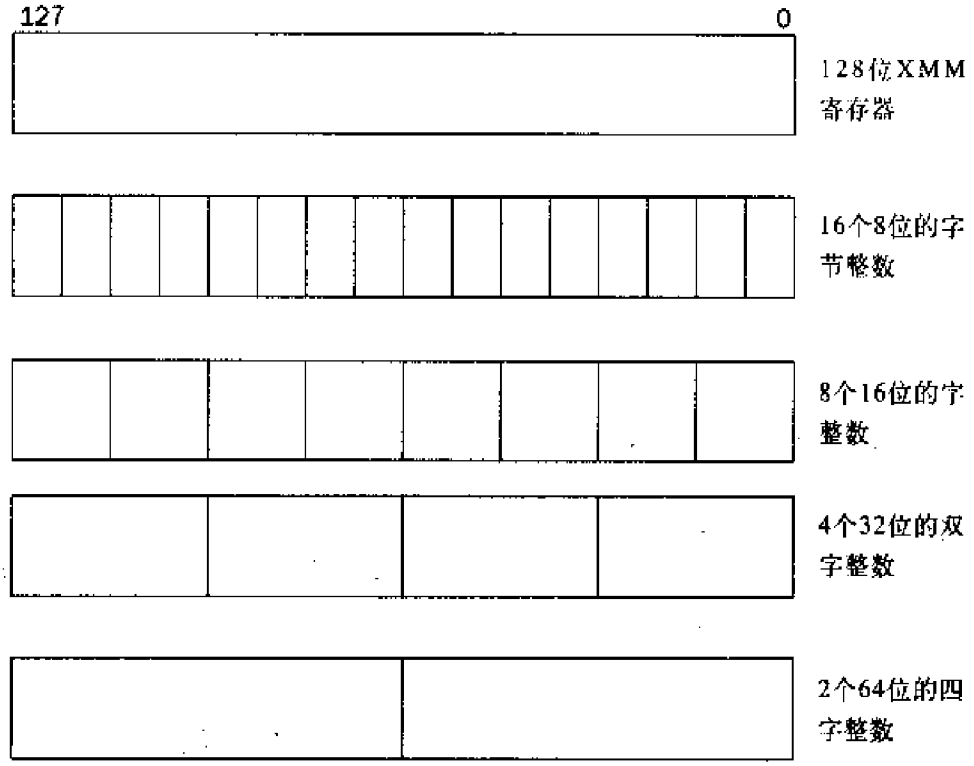



图 7-7

如图7-7所示，可以把16个字节整数、8个字整数、4个双字整数或者2个四字整数打包在单一128位SSE寄存器中。SSE平台提供附加的指令，用于对SSE寄存器中的打包数据值执行并行的数学操作。这使处理器可以使用相同的时钟周期处理多得多的信息。

### 7.3.4 传送SSE整数

MOVDQA和MOVDQU指令用于把128位数据传送到XMM寄存器中，或者在XMM寄存器之间传送数据。助记符的A和U部分代表对准和不对准，它们表示数据是如何存储在内存中的。对于对准16个字节边界的数据，就使用A选项；否则，就使用U选项（第5章“传送数据”介绍对准的数据）。

MOVDQA和MOVDQU指令的格式如下：

```
movdqa source, destination
```

其中source和destination可以是SSE 128位寄存器或者128位的内存位置（但是再次强调，不能在两个内存位置之间传送数据）。当使用对准的数据时，SSE指令执行得更快。还有，如果程

序对未对准的数据使用MOVDQA指令，就会造成硬件异常。

ssetest.s程序演示把128位数据传送到SSE寄存器中：

```
# ssetest.s - An example of using 128-bit SSE registers
.section .data
values1:
   .int 1, -1, 0, 135246
values2:
   .quad 1, -1
.section .text
.globl _start
_start:
   nop
   movdqu values1, %xmm0
   movdqu values2, %xmm1

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

ssetest.s程序定义两个包含不同整数数据类型的数据数组。values1数组包含4个双字带符号整数值，values2数组包含2个四字带符号整数值。使用MOVDQU指令把这两个数据数组传送到SSE寄存器中。

对程序进行汇编之后，可以在调试器中查看程序的结果。调试器能够使用print命令显示SSE寄存器（XMM0到XMM7）的值：

```
(gdb) print $xmm0
$1 = {v4_float = {1.40129846e-45, -nan(0x7fffff), 0, 1.89520012e-40},
  v2_double = {-nan(0xfffff00000001), 2.8699144274488922e-309},
  v16_int8 = "\001\000\000\000ÿÿÿÿ\000\000\000\000N\020\002", v8_int16 = {1,
    0, -1, -1, 0, 0, 4174, 2}, v4_int32 = {1, -1, 0, 135246}, v2_int64 = {
    -4294967295, 580877146914816},
  uint128 = 0x0002104e00000000ffffffff00000001}
(gdb) print $xmm1
$2 = {v4_float = {1.40129846e-45, 0, -nan(0x7fffff), -nan(0x7fffff)},
  v2_double = {4.9406564584124654e-324, -nan(0xfffffffffffff)},
  v16_int8 = "\001\000\000\000\000\000\000\000ÿÿÿÿÿÿÿÿ", v8_int16 = {1, 0, 0,
    0, -1, -1, -1, -1}, v4_int32 = {1, 0, -1, -1}, v2_int64 = {1, -1},
  uint128 = 0xffffffffffffffff0000000000000001}
(gdb)
```

MOVDQU指令执行之后，XMM0和XMM1寄存器包含数据段中定义的数据值。XMM0寄存器包含4个双字带符号整数数据值，XMM1寄存器包含2个四字带符号整数数据值。

记住，ssetest.s程序只能运行在奔腾III或者更新的处理器上。第17章“使用高级IA-32特性”将介绍MMX和SSE指令集以及如何使用它们。

## 7.4 二进制编码的十进制

二进制编码的十进制（Binary Coded Decimal，BCD）数据类型在计算机系统中已经存在很久了。BCD格式经常用于简化对使用十进制数字的设备（比如必须向人显示数字的设备，如时钟和计时器）的处理。处理器不是把十进制数字转换为二进制数字以便进行数学操作，然后再

转换回十进制；而是可以按照BCD格式保存数字并且执行数学操作。了解BCD如何工作以及处理器如何使用它，能对汇编语言程序设计有所帮助。下面几节介绍BCD格式和IA-32平台如何处理BCD数据类型。

### 7.4.1　BCD是什么

BCD的名称就说明了它的作用，它按照二进制格式对十进制数字进行编码。每个BCD值都是一个无符号8位整数，值的范围是0到9。在BCD中，大于9的8位值被认为是非法的。包含BCD值的字节组合在一起表示十进制的数位。在多字节的BCD值中，最低的字节保存十进制的个位的值，下一个较高位字节保存十位的值，依此类推。图7-8演示这种情况。

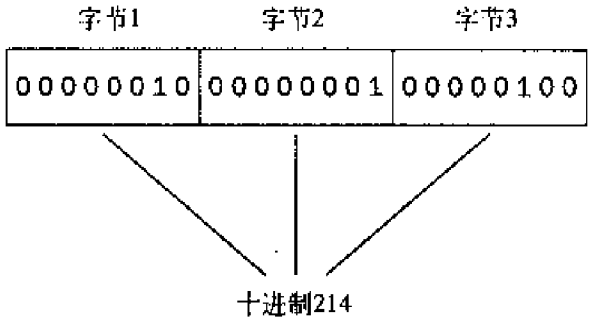



图　7-8

在图7-8中，十进制值214被表示为BCD值00000010 00000001 00000100。高位字节保存百位的值（2），下一个字节保存十位的值（1），最低的8位保存个位的值（4）。

读者能够发现，BCD使用整个字节表示每个十进制数位，这样浪费了空间。打包的BCD被创建出来，帮助弥补这一损失。打包的BCD允许单一字节包含两个BCD值。字节的低4位包含低位的BCD值，字节的高4位包含高位的BCD值。图7-9演示这种情况。

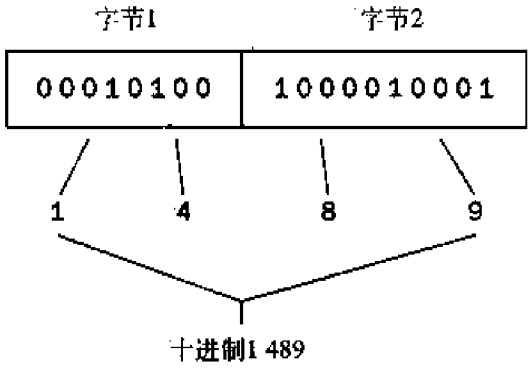



图　7-9

在图7-9中，十进制值1 489存储在2个字节的BCD值中。第一个字节包含前两个十进制数位（1和4），第二个字节包含后两个十进制数位（8和9）。

可以看到，即使是打包的BCD，效率也不高。使用4个字节，打包的BCD只能表示从0到9 999的数字。在无符号整数值中，使用4个字节可以表示的最大值是4 292 967 295。

从这些例子可以看出，IA-32平台中一般的BCD格式只支持无符号整数值。但是IA-32的FPU提供了支持带符号BCD整数的方法。

### 7.4.2　FPU BCD值

FPU寄存器可以用于在FPU之内进行BCD数学运算操作。FPU包含8个80位寄存器（从ST0到ST7），也可以使用它们保存80位BCD值。使用低位的9个字节存储BCD值，格式是打包BCD，每个字节包含两个BCD值（产生18个BCD数位）。大多数情况下，除了最高位的一位之外，不使用FPU寄存器的最高字节。这一位用作符号指示符——1表示负的BCD值，0表示正值。图7-10显示这一格式。

这一描述有些容易使人误解。为了把BCD值加载到FPU寄存器，必须使用80位打包BCD格

式在内存中创建值。值被传送给FPU寄存器之后，它就被自动地转换为扩展双精度浮点格式（参见本章后面7.5节）。在FPU中，对数据进行的任何数学操作都是按照浮点格式进行的。当准备从FPU获取结果时，浮点值被自动转换为80位打包BCD格式。

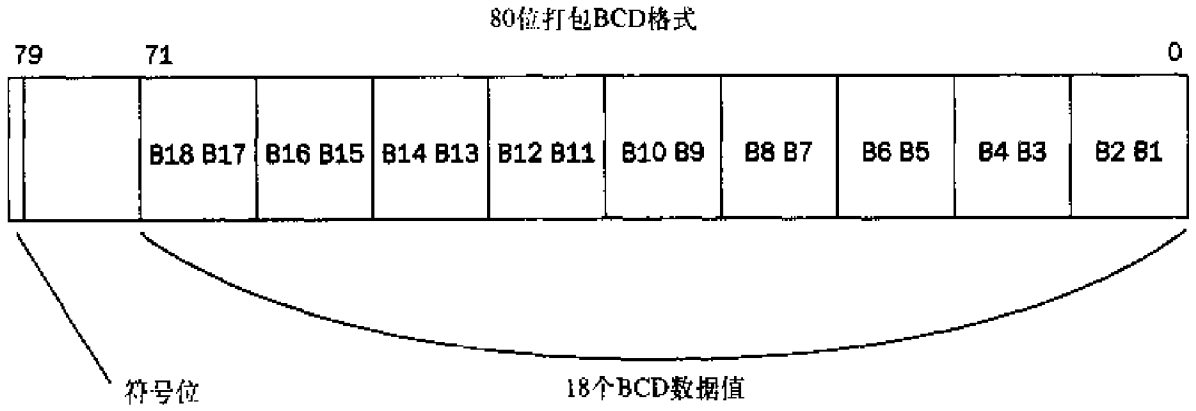



图 7-10

### 7.4.3 传送BCD值

IA-32指令集包含处理80位打包BCD值的指令。可以使用FBLD和FBSTP指令把80位打包BCD值加载到FPU寄存器中以及从FPU寄存器获取这些值。

使用FPU寄存器的方式和使用通用寄存器稍微有些区别。8个FPU寄存器的行为类似于内存中的堆栈区域。可以把值压入和弹出FPU寄存器池。ST0引用位于堆栈顶部的寄存器。当值被压入FPU寄存器堆栈时，它被存放在ST0寄存器中，ST0中原来的值被加载到ST1中。

*第9章“高级数学函数”中将更加详细地介绍FPU寄存器如何工作。*

FBLD指令用于把打包80位BCD值传送到FPU寄存器堆栈中。它的格式很简单：

```
fbld source
```

其中source是80位的内存位置。

bcdtest.s程序用于演示把BCD值加载到FPU寄存器中以及从FPU寄存器获取BCD值的基本操作：

```
# bcdtest.s - An example of using BCD integer values
.section .data
data1:
   .byte 0x34, 0x12, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00
data2:
   .int 2
.section .text
.globl _start
_start:
   nop
   fbld data1
   fimul data2
   fbstp data1

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

bcdtest.s程序在标签data1定义的内存位置创建一个表示十进制值1 234的简单的BCD值（记住Intel使用小尾数表示法）。使用FBLD指令把这个值加载到FPU寄存器堆栈的顶部（ST0）。使用FIMUL指令（在第9章中讨论）把ST0寄存器和data2所在的内存位置中的整数值相乘。最后，使用FBSTP指令把堆栈中新的值传送回data1所在的内存位置中。

对程序进行汇编之后，可以在调试器中运行它并且监视程序中不同位置的情况。首先，在执行任何指令之前，查看data1所在的内存位置的值：

```
(gdb) x/10b &data1
0x8049094 <data1>:   0x34    0x12    0x00    0x00    0x00    0x00    0x00    0x00
0x804909c <data1+8>: 0x00    0x00
(gdb)
```

很好。1 234的BCD值被加载到了data1所在的内存位置。下一步，单步执行FBLD指令，并且使用info all命令查看ST0寄存器中的值：

```
(gdb) s
12          fimul data2
(gdb) info all
.
.
st0            1234     (raw 0x40099a40000000000000)
```

当在寄存器列表中找到ST0寄存器的值时，应该显示它加载了十进制值1 234。但是读者也许会注意到，这个寄存器的十六进制值不是80位打包BCD格式。记住，在FPU中，BCD值被转换成了浮点表示方式。

现在单步执行下一条指令（FIMUL），并且再次查看寄存器：

```
(gdb) s
13          fbstp data1
(gdb) info all
.
.
st0            2468     (raw 0x400a9a40000000000000)
```

的确，ST0寄存器中的值和2相乘了。最后一个步骤应该把ST0中的值存放回data1所在的内存位置中。通过显示这个内存位置，可以检查操作的完成情况：

```
(gdb) x/10b &data1
0x8049094 <data1>:      0x68    0x24    0x00    0x00    0x00    0x00    0x00    0x00
0x804909c <data1+8>:    0x00    0x00
(gdb)
```

正如我们期望的，新的值被存放到了data1所在的内存位置，并且转换回了BCD格式。

第9章更加详细地讲解如何在数学运算操作中使用BCD值。

## 7.5　浮点数

既然读者已经了解了所有关于整数的知识，现在就应该开始转移到更加复杂的数字数据类型——浮点数的学习了。过去，整数的处理比较容易，因为Intel处理器总是包含对执行整数数

学操作的内置支持。在早期的Intel处理器（比如80286和80386芯片）中，执行浮点操作要么需要通过软件使用整数模拟浮点值，要么需要购买只用于专门执行浮点运算的单独的FPU芯片。

但是，从80486处理器开始，Intel的IA-32平台直接支持浮点操作。对于汇编语言程序员来说，现在把浮点数学操作并入程序中是很容易的工作。

本节介绍浮点数据类型是什么，并且演示如何在汇编语言程序中使用它。

### 7.5.1 浮点数是什么

到目前为止，本章介绍的所有数字系统都和整数有关。整数表示用于计数的数字，比如1只狗、2只猫和10匹马。最终，负数的概念被包含在整数中，成为带符号整数数字系统。整数和BCD数据类型都只能包含整数值。

正如你所知，使用整数并不能描述所有数字关系。在某些时候，需要引入小数的概念。这意味着在两个整数值之间能够包含无限数量的值。除了整数之间的无限数量的值之外，在数字系统中还有无限数量的整数值。所有这些数字组合在一起称为实数（real number）。实数可以包含从负无穷大到正无穷大的任何数字值，这些值可以带有任意数量的小数位。72 326.224 576是实数的一个例子。

在计算机上处理实数是一个挑战，特别是数字还有很多不同的数量级。开发浮点格式是为了产生在计算机系统中表示实数的标准方法。

1. 浮点格式

浮点格式使用科学记数法表示实数。如果读者在学校里学习过任何类型的科学课程，就可能熟悉科学记数法。科学记数法把数字表示为系数（coefficient）（也称为尾数（mantissa））和指数（exponent），比如3.684 5*10^2。在十进制领域中，指数的基数值为10，并且表示小数点移动多少位以生成系数。每次小数点向前移动时，指数就递增。每次小数点向后移动时，指数就递减。

例如，在科学记数法中，实数25.92可以表示为2.592*10^1。值2.592是系数，值10^1是指数。必须把系数和指数相乘，才能获得原始的实数。另一个例子是0.001 72可以表示为1.72*10^-3。数字1.72必须和10^-3相乘才能获得原始的值。

2. 二进制浮点格式

计算机系统使用二进制浮点数，这种格式使用二进制科学记数法格式表示值。因为数字按照二进制格式表示，所以系数和指数都基于二进制值，而不是十进制值。这种格式的一个例子如1.0101*2^2。处理系数的小数部分（小数点后面的部分）容易引起混乱。

为了对二进制浮点值进行译码，必须首先了解二进制小数数字的意义。在十进制领域中，读者习惯于看到0.159这样的值。这个值表示的是0 + (1/10) + (5/100) + (9/1000)。相同的原则也应用于二进制领域。

系数值1.0101乘以指数2^2应该生成二进制值101.01，它表示十进制整数5，加上分数(0/2) + (1/4)。这生成十进制值5.25。

二进制小数数字是处理浮点值的过程中最容易混淆的部分。下表列出前面几个二进制小数以及它们对应的十进制值。

| 二进制 | 十进制分数 | 十进制值 |
|---|---|---|
| 0.1 | 1/2 | 0.5 |
| 0.01 | 1/4 | 0.25 |
| 0.001 | 1/8 | 0.125 |
| 0.0001 | 1/16 | 0.0625 |
| 0.00001 | 1/32 | 0.03125 |
| 0.000001 | 1/64 | 0.015625 |

为了帮助说明二进制小数，下表列出了使用二进制浮点值的几个例子：

| 二进制 | 十进制分数 | 十进制值 |
|---|---|---|
| 10.101 | 2+1/2+1/8 | 2.625 |
| 10011.001 | 19+1/8 | 19.125 |
| 10110.1101 | 22+1/2+1/4+1/16 | 22.8125 |
| 1101.011 | 13+1/4+1/8 | 13.375 |

上表中的例子只具有有限的小数部分。但是，和十进制小数可能具有重复值（比如十进制值1/3）一样，二进制小数也可能具有重复的小数值。必须在某个位置截断这些值，并且只能以二进制格式估计十进制小数。

幸运的是，GNU汇编器会替我们完成这一工作，如果读者感觉没有充分理解二进制小数和二进制浮点格式，那也不必过于担心。

编写二进制浮点值时，二进制值通常被规格化了。这个操作把小数点移动到最左侧的数位并且修改指数进行补偿。例如值1101.011变成了1.101011*2^3。

在计算机时代的早期，试图在计算机系统中正确地表示二进制浮点数是一个挑战。幸运的是，标准被开发出来帮助程序员处理浮点数。一系列标准的浮点数据类型被创建出来，这简化了在计算机程序中处理实数的工作。下一节介绍标准浮点数据类型。

### 7.5.2 标准浮点数据类型

虽然可能的实数值的数量是无限的，但是处理器用来处理值的位的数量是有限的。出于这个原因，创建了一个标准系统用于在计算机环境中近似地表示实数。虽然近似方式并不完美，但是它们提供了处理现实的实数系统的子集的系统。

1985年，电气和电子工程师学会（Institute of Electrical and Electronics Engineers，IEEE）创建了称为IEEE标准754的浮点格式。这些格式用于在计算机系统中通用地表示实数。Intel在IA-32平台中采用这种标准来表示浮点值。

IEEE标准754浮点标准使用3个成分把实数定义为二进制浮点值：

- 符号
- 有效数字
- 指数

符号位表示值是负的还是正的。符号位中的1表示负值，0表示正值。

有效数字部分表示浮点数的系数（coefficient）（或者说尾数（mantissa））。系数可以是规格

化的（normalized），也可以是非规格化的（denormalized）。当二进制值被规格化时，它写为小数点前有个1。指数被修改，表示移动了多少位才可以实现规格化（和科学记数法的方法类似）。这意味着在规格化的值中，有效数字永远由1和二进制小数构成。

指数表示浮点数的指数部分。因为指数值可以是正值，也可以是负值，所以通过一个偏差值对它进行置偏。这样确保指数字段只能是无符号正整数。这还限制了这种格式中可用的最小和最大指数值。二进制浮点数的一般格式如图7-11所示。

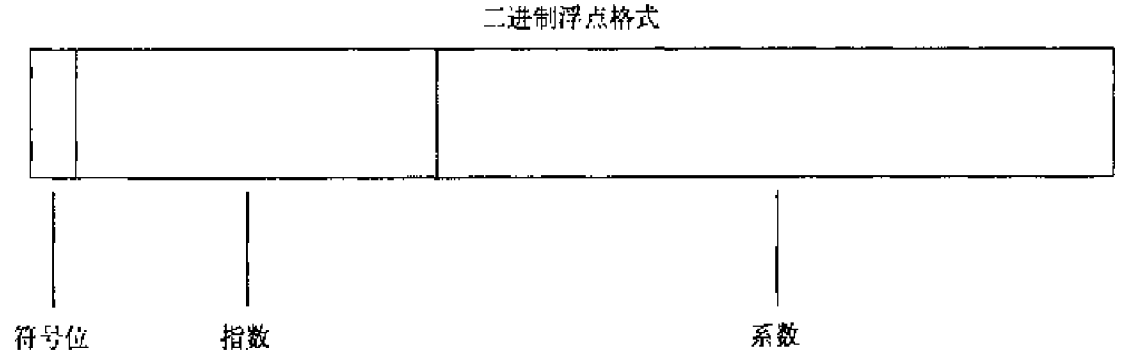



图 7-11

浮点数的这3个部分被包含在固定长度的数据格式之内。IEEE标准754定义了浮点数的两种长度：

- 32位（称为单精度）
- 64位（称为双精度）

可以用于表示有效数字的位的数量决定精度。图7-12显示两种不同精度类型的位的布局。

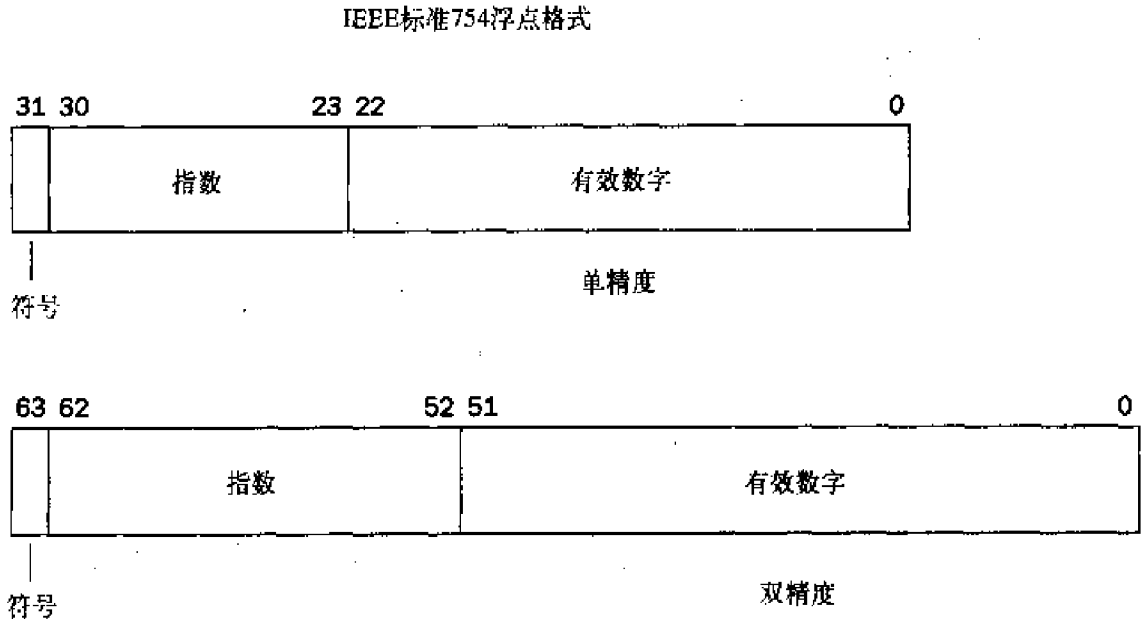



图 7-12

单精度浮点数使用23位有效数字值。但是，浮点格式假设有效数字的整数值永远是1，并且不在有效数字值中使用它。这样实际上使有效数字的精度达到了24位。指数使用8位值，偏差值为127。这就是说指数值的范围是−128到+127（二进制指数）。这种组合生成的单精度浮点数的

十进制范围是1.18*10^-38到3.40*10^38。

双精度浮点数使用52位小数值，它提供有效数字的精度为53位。指数使用11位值，偏差值为1 023。这就是说指数值的范围是-1 022到+1 023（二进制指数）。这种组合生成的双精度浮点数的十进制范围是2.23*10^-308到1.79*10^308。

### 7.5.3 IA-32浮点值

IA-32平台使用IEEE标准754的单精度和双精度浮点格式，还使用它自己的80位格式，称为扩展双精度浮点格式。在执行浮点运算时，这3种格式提供不同的精度级别。在浮点运算处理的过程中，扩展双精度浮点格式使用在80位FPU寄存器内。

Intel的80位扩展双精度浮点格式使用64位作为有效数字，使用15位作为指数。扩展双精度浮点格式使用的偏差值为16 383，生成的指数范围是-16 382到+16 383，相应的十进制范围是3.37*10^-4932到1.18*10^4932。

下表总结在标准IA-32平台上使用的3种类型的浮点格式。

| 数据类型 | 长度 | 有效数字位 | 指数位 | 范　围 |
|---|---|---|---|---|
| 单精度 | 32 | 24 | 8 | 1.18*10^-38到3.40*10^38 |
| 双精度 | 64 | 53 | 11 | 2.23*10^-308到1.79*10^308 |
| 扩展双精度 | 80 | 64 | 15 | 3.37*10^-4932到1.18*10^4932 |

### 7.5.4 在GNU汇编器中定义浮点值

GNU汇编器（gas）提供了定义单精度和双精度浮点值的命令（参见第5章）。到编写本书时，gas还没有定义扩展双精度浮点值的命令。

浮点值按照小尾数格式存储在内存中。数组按照命令中定义值的顺序存储。.float命令用于创建32位单精度值，.double命令用于创建64位双精度值。

### 7.5.5 传送浮点值

FLD指令用于把浮点值传送入和传送出FPU寄存器。FLD指令的格式是：

```
fld source
```

其中source可以是32位、64位或者80位内存位置。

floattest.s程序演示如何在汇编语言程序中定义和使用浮点数据值：

```
# floattest.s - An example of using floating point numbers
.section .data

value1:
   .float 12.34
value2:
   .double 2353.631
.section .bss
   .lcomm data, 8
.section .text
```

```
.globl _start
_start:
   nop
   flds value1
   fldl value2
   fstl data

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

标签value1指向存储在4个字节内存中的单精度浮点值。标签value2指向存储在8个字节内存中的双精度浮点值。标签data指向内存中的空缓冲区，它将被用于传输双精度浮点值。

IA-32的FLD指令用于把存储在内存中的单精度和双精度浮点数加载到FPU寄存器堆栈中。为了区分这两种数据长度，GNU汇编器使用FLDS指令加载单精度浮点数，而使用FLDL指令加载双精度浮点数。

类似地，FST指令用于获取FPU寄存器堆栈中顶部的值，并且把这个值存放到内存位置中。对于单精度数字，使用的指令是FSTS，双精度数字使用的指令是FSTL。

对floattest.s程序进行汇编之后，在执行指令时监视内存位置和寄存器值。首先，查看浮点值是如何存储在内存位置中的：

```
(gdb) x/4b &value1
0x8049094 <value1>: 0xa4    0x70    0x45    0x41
(gdb) x/8b &value2
0x8049098 <value2>: 0x8d    0x97    0x6e    0x12    0x43    0x63    0xa2    0x40
(gdb)
```

如果希望查看十进制值，可以使用x命令的f选项：

```
(gdb) x/f &value1
0x8049094 <value1>: 12.3400002
(gdb) x/gf &value2
0x8049098 <value2>: 2353.6309999999999
(gdb)
```

注意，当调试器试图计算要显示的值时，舍入错误已经存在。f选项只能显示单精度数字。为了显示双精度值，需要使用gf选项，它显示四字值。

单步运行第一条FLDS指令之后，使用info reg或者print命令查看ST0寄存器的值：

```
(gdb) print $st0
$1 = 12.340000152587890625
(gdb)
```

位于value1内存位置中的值被正确地存放到了ST0寄存器中。现在单步运行下一条指令，并且查看ST0寄存器中的值：

```
(gdb) print $st0
$2 = 2353.6309999999998581188265234231949
(gdb)
```

这个值已经被替换为新加载的双精度值（并且调试器正确地把这个值显示为双精度浮点数）。

为了查看对原来加载的值进行了什么处理，查看ST1寄存器：

```
(gdb) print $st1
$3 = 12.340000152587890625
(gdb)
```

正如我们期望的，当加载新的值时，ST0中的值被下移到了ST1寄存器中。现在查看data标签的值，单步执行FSTL指令，并且再次查看：

```
(gdb) x/gf &data
0x80490a0 <data>:       0
(gdb) s
18          movl $1, %eax
(gdb) x/gf &data
0x80490a0 <data>:       2353.6309999999999
(gdb)
```

FSTL指令把ST0寄存器中的值加载到了data标签指向的内存位置中。

### 7.5.6 使用预置的浮点值

IA-32指令集包含一些预置的浮点值，可以把它们加载到FPU寄存器堆栈中。下表列出了这些指令。

| 指　令 | 描　述 |
|---|---|
| FLD1 | 把+1.0压入FPU堆栈中 |
| FLDL2T | 把10的对数（底数2）压入FPU堆栈中 |
| FLDL2E | 把e的对数（底数2）压入FPU堆栈中 |
| FLDPI | 把pi的值压入FPU堆栈中 |
| FLDLG2 | 把2的对数（底数10）压入FPU堆栈中 |
| FLDLN2 | 把2的对数（底数e）压入FPU堆栈中 |
| FLDZ | 把+0.0压入FPU堆栈中 |

这些指令提供把常用数学值压入FPU堆栈的简便方式，以便对数据进行操作。读者也许会注意到FLDZ指令有些奇怪。在浮点数据类型中，+0.0和-0.0之间是有区别的。对于大多数操作，它们被认为是相同的值，但是使用在除法中时，它们产生不同的值（正无穷大和负无穷大）。

fpuvals.s程序演示如何使用预置的浮点值：

```
# fpuvals.s - An example of pushing floating point constants
.section .text
.globl _start
_start:
   nop
   fld1
   fldl2t
   fldl2e
   fldpi
   fldlg2
   fldln2
   fldz
```

```
movl $1, %eax
movl $0, %ebx
int $0x80
```

fpuvals.s程序简单地把各个浮点常量压入到FPU寄存器堆栈中。可以对程序进行汇编并且在调试器中运行它，从而在执行指令时监视FPU寄存器堆栈。在清单的结尾，寄存器应该如下：

```
(gdb) info all
.
.
.
st0   0          (raw 0x00000000000000000000)
st1   0.693147180559945309428690474184975 3   (raw 0x3ffeb17217f7d1cf79ac)
st2   0.301029995663981195225646428359489 45  (raw 0x3ffd9a209a84fbcff799)
st3   3.141592653589793238512808959406186 2   (raw 0x4000c90fdaa22168c235)
st4   1.442695040888963407387651782798343 4   (raw 0x3fffb8aa3b295c17f0bc)
st5   3.321928094887362347808340556909456 6   (raw 0x4000d49a784bcd1b8afe)
st6   1          (raw 0x3fff8000000000000000)
st7   0          (raw 0x00000000000000000000)
(gdb)
```

值的顺序和它们被存放到堆栈中的顺序是相反的。

### 7.5.7 SSE浮点数据类型

除了3种标准浮点数据类型之外，实现SSE技术的Intel处理器还包含两种高级浮点数据类型。SSE技术引入了8个128位XMM寄存器（更多信息参见第2章），可以使用这些寄存器保存打包浮点数。

和打包BCD的概念类似，打包浮点数使多个浮点值可以存储在单一寄存器中。可以使用多个数据元素并行地执行浮点计算，这会比串行处理数据更快地生成结果。

下面是可用的2种新的128位浮点数据类型：

- 128位打包单精度浮点（SSE中）
- 128位打包双精度浮点（SSE2中）

因为一个单精度浮点值需要32位，所以128位寄存器可以保存4个打包单精度浮点值。类似地，它可以保存2个64位打包双精度浮点值。如图7-13所示。

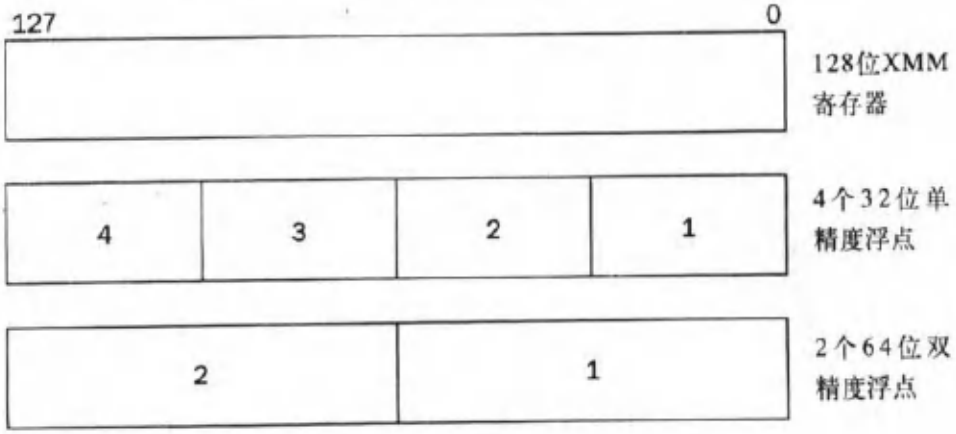



图 7-13

这些新的数据类型在FPU或者MMX寄存器中是不可用的。只能在XMM寄存器中使用它们，并且只能在支持SSE或者SSE2的处理器上使用。必须使用专门的指令加载和获取数据值，同样必须使用专门的数学指令对打包浮点数据进行数学操作。

### 7.5.8 传送SSE浮点值

正如我们期望的，IA-32指令集包含用于在处理器中传送新的SSE浮点数据类型值的指令。这些指令分为对打包单精度浮点数据进行操作的SSE指令，以及对打包双精度浮点数据进行操作的SSE2指令。

1. SSE浮点值

有一个完整的指令集用于在内存和处理器上的XMM寄存器之间传送128位打包单精度浮点值。下表列出用于传送SSE打包单精度浮点数据的指令。

| 指　令 | 描　述 |
|---|---|
| MOVAPS | 把4个对准的打包单精度值传送到XMM寄存器或者内存 |
| MOVUPS | 把4个不对准的打包单精度值传送到XMM寄存器或者内存 |
| MOVSS | 把1个单精度值传送到内存或者寄存器的低双字 |
| MOVLPS | 把2个单精度值传送到内存或者寄存器的低四字 |
| MOVHPS | 把2个单精度值传送到内存或者寄存器的高四字 |
| MOVLHPS | 把2个单精度值从低四字传送到高四字 |
| MOVHLPS | 把2个单精度值从高四字传送到低四字 |

这些指令的每一条都使用128位XMM寄存器在XMM寄存器和内存之间传送打包32位单精度浮点值。不仅可以传送整组的打包单精度浮点值，也可以在XMM寄存器之间传送2个打包单精度浮点值的子集。

ssefloat.s中显示的是传送SSE打包单精度浮点值的一个例子：

```
# ssefloat.s - An example of moving SSE FP data types
.section .data
value1:
   .float 12.34, 2345.543, -3493.2, 0.44901
value2:
   .float -5439.234, 32121.4, 1.0094, 0.000003
.section .bss
   .lcomm data, 16
.section .text
.globl _start
_start:
   nop
   movups value1, %xmm0
   movups value2, %xmm1
   movups %xmm0, %xmm2
   movups %xmm0, data

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

ssefloat.s程序创建2个数据数组（分别是value1和value2），每个数组由4个单精度浮点值组成。它们将成为被存储到XMM寄存器中的打包数据值。还创建了一个数据缓冲区，它有足够的空间保存4个单精度浮点值（即一个打包的值）。然后程序使用MOVUPS指令在XMM寄存器和内存之间传送打包单精度浮点值。

对程序进行汇编之后，可以在调试器中查看程序执行情况。单步执行前3个指令之后，XMM寄存器应该如下：

```
(gdb) print $xmm0
$1 = {v4_float = {5.84860315e+35, 2.63562489, 1.79352231e-36, 5.07264233},
  v2_double = {12.34, 2345.5430000000001},
  v16_int8 = "xxxx\024x(@u\223\030\004\026Sx@", v8_int16 = {18350, 31457,
    -20972, 16424, -27787, 1048, 21270, 16546}, v4_int32 = {2061584302,
    1076407828, 68719477, 1084379926}, v2_int64 = {4623136420479977390,
    4657376318677619573}, uint128 = 0x40a25316041893754028ae147ae147ae}
(gdb) print $xmm1
$2 = {v4_float = {-1.11704749e+24, -5.66396856, -1.58818684e-23, 6.98026705},
  v2_double = {-5439.2340000000004, 32121.400000000001},
  v16_int8 = "D\213lxxxx\232\231\231\231xxxx", v8_int16 = {-29884, -6292,
    16187, -16203, -26214, -26215, 24153, 16607}, v4_int32 = {-412316860,
    -1061863621, -1717986918, 1088380505}, v2_int64 = {-4560669521124488380,
    4674558677155944858}, uint128 = 0x40df5e599999999ac0b53f3be76c8b44}
(gdb) print $xmm2
$3 = {v4_float = {5.84860315e+35, 2.63562489, 1.79352231e-36, 5.07264233},
  v2_double = {12.34, 2345.5430000000001},
  v16_int8 = "xxxx\024x(@u\223\030\004\026Sx@", v8_int16 = {18350, 31457,
    -20972, 16424, -27787, 1048, 21270, 16546}, v4_int32 = {2061584302,
    1076407828, 68719477, 1084379926}, v2_int64 = {4623136420479977390,
    4657376318677619573}, uint128 = 0x40a25316041893754028ae147ae147ae}
(gdb)
```

从输出可以发现，所有数据都被正确地加载到了XMM寄存器中。v4_float格式显示使用的打包单精度浮点值。

最后的指令步骤是把XMM寄存器的值复制到data位置。可以使用x/4f命令显示结果：

```
(gdb) x/4f &data
0x80490c0 <data>:  12.3400002      2345.54297      -3493.19995      0.449010015
(gdb)
```

为了把存储在内存位置data中的字节显示为4个单精度浮点值，可以使用x命令的4f选项。这把8个字节解释为正确的格式。内存位置data现在包含从内存位置value1加载到XMM寄存器中，并且被复制到内存位置data的数据。为了防止调试器中的舍入错误的欺骗，可以按照十六进制格式复查答案：

```
(gdb) x/16b &data
0x80490c0 <data>:    0xa4    0x70    0x45    0x41    0xb0    0x98    0x12    0x45
0x80490c8 <data+8>:  0x33    0x53    0x5a    0xc5    0xa4    0xe4    0xe5    0x3e
(gdb) x/16b &value1
0x804909c <value1>:  0xa4    0x70    0x45    0x41    0xb0    0x98    0x12    0x45
0x80490a4 <value1+8>:0x33    0x53    0x5a    0xc5    0xa4    0xe4    0xe5    0x3e
(gdb)
```

的确，它们是匹配的！

2. SSE2浮点值

和SSE数据类型类似，IA-32平台包含用于传送新的SSE2打包双精度浮点数据类型的指令。下表列出可以使用的新指令。

| 指　令 | 描　述 |
|---|---|
| MOVAPD | 把2个对准的双精度值传送到XMM寄存器或者内存 |
| MOVUPD | 把2个不对准的双精度值传送到XMM寄存器或者内存 |
| MOVSD | 把1个双精度值传送到内存或者寄存器的低四字 |
| MOVHPD | 把1个双精度值传送到内存或者寄存器的高四字 |
| MOVLPD | 把1个双精度值传送到内存或者寄存器的低四字 |

这些指令的每一条都使用128位XMM寄存器传送64位双精度浮点值。MOVAPD和MOVUPD指令把完整的打包双精度浮点值传送到和传送出XMM寄存器。

sse2float.s程序演示这些指令:

```
# sse2float.s - An example of moving SSE2 FP data types
.section .data
value1:
   .double 12.34, 2345.543
value2:
   .double -5439.234, 32121.4
.section .bss
   .lcomm data, 16
.section .text
.globl _start
_start:
   nop
   movupd value1, %xmm0
   movupd value2, %xmm1
   movupd %xmm0, %xmm2
   movupd %xmm0, data

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

这一次，存储在内存中的数值被改为双精度浮点值。因为程序将传输打包值，所以创建了一个包含2个值的数组。

对程序进行汇编之后，可以再次在调试器中检查程序的操作。单步执行MOVUPD指令之后，查看有关的XMM寄存器的内容:

```
(gdb) print $xmm0
$1 = {v4_float = {0.0587499999, 2.57562494, -7.46297859e-36, -2.33312488},
  v2_double = {10.42, -5.3300000000000001},
  v16_int8 = "xxp=\xx$@Rx\036\205xQ\025x", v8_int16 = {-23593, 15728, -10486,
    16420, -18350, -31458, 20971, -16363}, v4_int32 = {1030792151, 1076156170,
    -2061584302, -1072344597}, v2_int64 = {4622055556569408471,
    -4605684971923916718}, uint128 = 0xc01551eb851eb8524024d70a3d70a3d7}
(gdb) print $xmm1
$2 = {v4_float = {0, 2.265625, -107374184, 2.01249981}, v2_double = {4.25,
    2.1000000000000001},
```

```
  v16_int8 = "\000\000\000\000\000\000\021@xxxxxx\000@", v8_int16 = {0, 0, 0,
    16401, -13107, -13108, -13108, 16384}, v4_int32 = {0, 1074855936,
    -858993459, 1073794252}, v2_int64 = {4616471093031469056,
    4611911198408756429}, uint128 = 0x4000cccccccccccd4011000000000000}
(gdb) print $xmm2
$3 = {v4_float = {1.40129846e-44, 2.80259693e-44, 4.20389539e-44,
    5.60519386e-44}, v2_double = {4.2439915824246103e-313,
    8.4879831653432862e-313},
  v16_int8 = "\n\000\000\000\024\000\000\000\036\000\000\000(\000\000",
  v8_int16 = {10, 0, 20, 0, 30, 0, 40, 0}, v4_int32 = {10, 20, 30, 40},
  v2_int64 = {85899345930, 171798691870},
  uint128 = 0x0000002800000001e000000140000000a}
(gdb) print $xmm3
$4 = {v4_float = {7.00649232e-45, 2.1019477e-44, 3.50324616e-44,
    4.90454463e-44}, v2_double = {3.1829936866949413e-313,
    7.4269852696136172e-313},
  v16_int8 = "\005\000\000\000\017\000\000\000\031\000\000\000#\000\000",
  v8_int16 = {5, 0, 15, 0, 25, 0, 35, 0}, v4_int32 = {5, 15, 25, 35},
  v2_int64 = {64424509445, 150323855385},
  uint128 = 0x0000002300000019000000f00000005}
(gdb)
```

同样，必须查看调试器的输出，但是这一次寻找的是v2_double数据类型。正确的值已经被传送到了寄存器中。

下一步，检查内存位置data，确认正确的值也被复制到了这里：

```
(gdb) x/2gf &data
0x80490c0 <data>:       12.34   2345.5430000000001
(gdb)
```

因为内存位置data包含2个双精度浮点值，所以必须使用x命令的2gf选项显示存储在这个内存位置的2个值。同样，我们得到了期望的结果。

3. SSE3指令

最后介绍奔腾4和更新的处理器上可用的SSE3技术，它添加了3个附加指令来帮助传送打包双精度浮 点值：

- MOVSHDUP：从内存或者XMM寄存器传送128位值，复制第2个和第4个32位数据元素。因此，传送由32位单精度浮点值DCBA构成的数据元素将创建由DDBB构成的128位打包单精度浮点值。
- MOVSLDUP：从内存或者XMM寄存器传送128位值，复制第1个和第3个32位数据元素。因此，传送由32位单精度浮点值DCBA构成的数据元素将创建由CCAA构成的128位打包单精度浮点值。
- MOVDDUP：从内存或者XMM寄存器传送64位双精度浮点值，把它复制到128位XMM寄存器中。因此，传送由64位双精度浮点值A构成的数据元素将创建128位打包双精度浮点值AA。

到编写本书时，支持SSE3指令的IA-32处理器只有奔腾4超线程（Hyperthreading）处理器。

## 7.6 转换

IA-32指令集包含众多指令，用于把以一种数据类型表示的数据转换为另一种数据类型。程

序需要把浮点数据转换为整数值（或者相反的转换）的情况并不少见。这些指令提供完成这种操作的简便方式，无需编写自己的算法。

### 7.6.1 转换指令

有很多指令用于转换数据类型，因为有不同的数据类型需要进行相互转换。下表介绍转换指令。

| 指　令 | 转　换 |
|---|---|
| CVTDQ2PD | 打包双字整数到打包双精度FP（XMM） |
| CVTDQ2PS | 打包双字整数到打包单精度FP（XMM） |
| CVTPD2DQ | 打包双精度FP到打包双字整数（XMM） |
| CVTPD2PI | 打包双精度FP到打包双字整数（MMX） |
| CVTPD2PS | 打包双精度FP到打包单精度FP（XMM） |
| CVTPI2PD | 打包双字整数到打包双精度FP（XMM） |
| CVTPI2PS | 打包双字整数到打包单精度FP（XMM） |
| CVTPS2DQ | 打包单精度FP到打包双字整数（XMM） |
| CVTPS2PD | 打包单精度FP到打包双精度FP（XMM） |
| CVTPS2PI | 打包单精度FP到打包双字整数（MMX） |
| CVTTPD2PI | 打包双精度FP到打包双字整数（MMX，截断） |
| CVTTPD2DQ | 打包双精度FP到打包双字整数（XMM，截断） |
| CVTTPS2DQ | 打包单精度FP到打包双字整数（XMM，截断） |
| CVTTPS2PI | 打包单精度FP到打包双字整数（MMX，截断） |

上表中的描述涉及可以在其中存放结果的目标寄存器，目标寄存器可以是MMX寄存器，也可以是XMM寄存器。另外，最后4条指令是截断的转换。在其他指令中，如果转换不精确，就会由XMM MXCSR寄存器的13位和14位控制进行舍入。这些位确定值被向上入还是向下舍。在截断的转换中，会自动执行向零方向的舍入。

源值可以从内存位置、MMX寄存器（对于64位值）或者XMM寄存器（对于64位或者128位值）获得。

### 7.6.2 转换范例

convtest.s程序是转换数据类型的一个例子：

```
# convtest.s - An example of data conversion
.section .data
value1:
   .float 1.25, 124.79, 200.0, -312.5
value2:
   .int 1, -435, 0, -25
.section .bss
data:
   .lcomm data, 16
.section .text
.globl _start
_start:
   nop
```

```
    cvtps2dq value1, %xmm0
    cvttps2dq value1, %xmm1
    cvtdq2ps value2, %xmm2
    movdqu %xmm0, data

    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

convtest.s程序在内存位置value1定义一个打包单精度浮点值，在内存位置value2定义一个打包双字整数值。第一对指令可以比较CVTPS2DQ和CVTTPS2DQ指令的结果。第一条指令执行一般的舍入，第二条指令通过向零方向舍入进行截断。

进行通常的汇编和调试之后，可以通过查看XMM寄存器和data内存位置来查看转换指令是如何操作的。通过查看XMM0和XMM1寄存器，可以发现截断造成的差异：

```
(gdb) print $xmm0
$1 = {v4_float = {1.40129846e-45, 1.75162308e-43, 2.80259693e-43,
    -nan(0x7ffec8)}, v2_double = {2.6524947387115311e-312,
    -nan(0xffec8000000c8)},
  v16_int8 = "\001\000\000\000}\000\000\000x\000\000\000xxxx", v8_int16 = {1,
    0, 125, 0, 200, 0, -312, -1}, v4_int32 = {1, 125, 200, -312}, v2_int64 = {
    536870912001, -1340029796152},
  uint128 = 0xfffffec8000000c80000007d00000001}
(gdb) print $xmm1
$2 = {v4_float = {1.40129846e-45, 1.7376101e-43, 2.80259693e-43,
    -nan(0x7ffec8)}, v2_double = {2.6312747808018783e-312,
    -nan(0xffec8000000c8)},
  v16_int8 = "\001\000\000\000|\000\000\000x\000\000\000xxxx", v8_int16 = {1,
    0, 124, 0, 200, 0, -312, -1}, v4_int32 = {1, 124, 200, -312}, v2_int64 = {
    532575944705, -1340029796152},
  uint128 = 0xfffffec8000000c80000007c00000001}
(gdb)
```

按照v4_int32格式，值被正确地显示出来。正如所见，一般转换把浮点值124.79舍入为125，但是截断转换把它向零方向舍入，使之成为124。内存位置data的值被转换为打包双字整数后，可以使用x/4d命令显示它：

```
(gdb) x/4d &data
0x80490c0 <data>: 1       125     200     -312
(gdb)
```

正如我们期望的，显示出了舍入后的整数值。

## 7.7 小结

本章介绍了很多关于IA-32平台如何处理数字的背景知识。最基本的数字数据类型是整数，用于表示整数值。支持的整数类型有2种：无符号整数，它只允许正值；和带符号整数，它包含正值和负值。IA-32平台支持4种长度的整数——字节、字、双字和四字。只要使用正确的格式进行扩展，以一种长度表示的整数数据类型可以扩展为更大的长度，而且不会丢失值。

Intel的SIMD技术引入了附加的整数数据类型。打包的字节和双字整数类型使多个带符号整

数值可以被包含在单一128位寄存器中。有专门的指令可以对多个整数值并行地执行数学操作（比如加法和减法），这帮助提高应用程序的执行速度。

二进制编码的十进制（Binary Coded Decimal，BCD）数据类型可以使用可读的十进制格式存储和使用整数。每个字节表示值中的单个十进制数位。因为一个十进制数值只需要4位，所以通过允许把2个十进制数位保存在单个字节中，可以使用打包BCD降低空间的需求。

浮点数据类型是最有用的，但是可能也是最复杂的数据类型。浮点数据类型用于在程序中表示实数。IA-32平台使用IEEE标准754格式定义单精度（32位）和双精度（64位）浮点值。第3种格式是扩展双精度浮点，它使用80位FPU寄存器执行更加精确的浮点运算。

SSE平台（在奔腾III处理器中出现）和SSE2平台（在奔腾4处理器中出现）也增加了新的数据类型。SSE平台引入了80位打包的字节和双字整数值，SSE2平台引入了128位打包的字节、字、双字和四字整数，以及128位打包单精度和双精度浮点值。

最后，本章讨论了可以用于在不同格式之间转换数字数据类型的IA-32指令。当需要把浮点值舍入为整数值时，或者对整数值执行浮点数学操作时（比如除法），这些指令会有所帮助。

既然读者已经了解了关于数字数据类型的所有知识，现在是开始进行一些数学操作的时候了。下一章研究IA-32平台的数学功能，并且演示如何使用不同的数据类型进行不同的数学运算。

# 第 8 章　基本数学功能

既然读者已经具备了不同数字数据类型的知识，可以开始在数学操作中使用它们。本章研究汇编语言中基本整数数学功能。

本章首先介绍整数运算是如何在处理器上执行的，以及可以如何在汇编语言程序中使用它们。本章介绍用于无符号和带符号数字的加法、减法、乘法和除法的指令，以及如何在程序中使用它们的范例。接下来，介绍移位指令，它们能够帮助提高乘法和除法操作的性能。然后，讲解十进制运算，还有实现十进制运算的指令。本章最后介绍IA-32平台上可用的基本布尔逻辑和位测试指令。

## 8.1　整数运算

汇编语言程序中执行数学操作的基本过程是整数运算。在读者试图研究更加复杂的浮点数学功能之前，应该完整地了解处理器如何对整数执行数学操作。

本节介绍如何在处理器上执行整数运算，以及可以如何在汇编语言程序中使用整数运算。这包括整数加法、减法、乘法和除法。

### 8.1.1　加法

似乎把两个整数加在一起是简单明了的过程，但是情况并不总是如此。需要预见在前进的道路上有一些障碍，它们都和二进制数值如何相加有关。下面几节介绍用于整数加法的指令。

1. ADD指令

ADD指令用于把两个整数相加。ADD指令的格式如下：

```
add source, destination
```

其中source可以是立即值、内存位置或者寄存器。destination参数可以是寄存器或者内存位置中存储的值（但是不能同时使用内存位置作为源和目标）。加法的结果存放在目标位置。

ADD指令可以将8位、16位或者32位值相加。和其他GNU汇编器指令一样，必须通过在ADD助记符的结尾添加b（用于字节）、w（用于字）或者l（用于双字）来指定操作数的长度。下面是使用ADD指令的一些例子：

```
addb $10, %al  # adds the immediate value 10 to the 8-bit AL register
addw %bx, %cx  # adds the 16-bit value of the BX register to the CX register
addl data, %eax   # adds the 32-bit integer value at the data label to EAX
addl %eax, %eax   # adds the value of the EAX register to itself
```

addtest1.s程序中演示这些指令：

```
# addtest1.s - An example of the ADD instruction
.section .data
data:
   .int 40
```

```
.section .text
.globl _start
_start:
   nop

   movl $0, %eax
   movl $0, %ebx
   movl $0, %ecx
   movb $20, %al
   addb $10, %al
   movsx %al, %eax
   movw $100, %cx
   addw %cx, %bx
   movsx %bx, %ebx
   movl $100, %edx
   addl %edx, %edx
   addl data, %eax
   addl %eax, data
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

如果没有使用整个32位寄存器，就要确保使用零填充目标寄存器，使寄存器的高位中没有内容，这总是个好主意。可以使用XOR指令轻易地完成这一操作（参见本章后面的8.4节）。确保目标寄存器被填充为零后，可以执行不同的ADD指令。注意，在能够正确地使用EAX寄存器的值之前，要使用MOVSX指令把AL寄存器的带符号整数值扩展到EAX寄存器，把BX的值扩展到EBX寄存器（关于扩展带符号整数的更多信息参见第7章“使用数字”）。

对程序进行汇编之后，可以单步执行指令并且在每条指令执行之后查看寄存器的值。运行完所有指令之后，应该得到如下结果：

```
(gdb) print $eax
$1 = 70
(gdb) print $ebx
$2 = 100
(gdb) print $ecx
$3 = 100
(gdb) print $edx
$4 = 200
(gdb) x/d &data
0x804909c <data>:       110
(gdb)
```

所有指令都按照期望的那样执行了。就像第7章中提到的，使用补码表示负数的优势在于可以使用相同的硬件执行带符号和无符号整数的加法。addtest2.s程序演示使用ADD指令执行一些带符号整数的加法：

```
# addtest2.s - An example of the ADD instruction and negative numbers
.section .data
data:
   .int -40
.section .text
.globl _start
_start:
   nop
```

```
movl $-10, %eax
movl $-200, %ebx
movl $80, %ecx
addl data, %eax
addl %ecx, %eax
addl %ebx, %eax
addl %eax, data
addl $210, data
movl $1, %eax
movl $0, %ebx
int $0x80
```

不管带符号整数值的符号是什么，ADD指令都能正确地执行加法操作。应该得到如下结果：

```
(gdb) print $eax
$1 = -170
(gdb) print $ebx
$2 = -200
(gdb) print $ecx
$3 = 80
(gdb) x/d &data
0x80490ac <data>:       0
(gdb)
```

ADD指令对程序使用的所有带符号整数正确地执行了加法操作。能够使用相同指令对带符号和无符号整数值进行加法操作是很方便的。

2. 检测进位或者溢出情况

当整数相加时，总是应该注意EFLAGS寄存器，以便确保操作过程中不会发生奇怪的事情。对于无符号整数，当二进制加法造成进位情况时（即结果大于允许的最大值），进位标志（carry flag）就会被设置为1。对于带符号整数，当出现溢出情况时（结果值小于允许的最小负值，或者大于允许的最大正值），溢出标志（overflow flag）就会被设置为1。当这些标志被设置为1时，就知道目标操作数的长度太小，不能保存加法的结果值，并且包含非法值。这个值将是答案的“溢出”了的部分。

进位和溢出标志的设置与加法中使用的数据长度相关联。例如，在ADDB指令中，如果结果超过255，就会把进位标志设置为1，但是在ADDW指令中，除非结果超过65 535，否则是不会设置进位标志的。

addtest3.s程序演示如何检测无符号整数加法中的进位情况：

```
# addtest3.s - An example of detecting a carry condition
.section .text
.globl _start
_start:
   nop
   movl $0, %ebx
   movb $190, %bl
   movb $100, %al
   addb %al, %bl
   jc over
   movl $1, %eax

   int $0x80
```

```
over:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

addtest3.s程序对存储在AL和BL寄存器中的2字节无符号整数值执行简单的加法。如果加法操作造成进位，则把进位标志设置为1，并且JC指令将跳转到标签over。程序的结果代码要么是加法的结果，要么就是0值（如果结果超过255）。因为我们设置了AL和BL寄存器中的值，所以我们可以控制程序中出现的情况。

测试这个程序是很容易的工作。首先，设置寄存器值使加法产生进位，运行程序，然后使用echo命令查看结果代码：

```
$ ./addtest3
$ echo $?
0
$
```

结果代码为0，表示正确地检测到了进位情况。现在，改动寄存器的值，使加法不产生进位：

```
movb $190, %bl
movb $10, %al
```

运行程序后，应该得到下面的结果：

```
$ ./addtest3
$ echo $?
200
$
```

加法没有产生进位，没有采取跳转，并且加法的结果被设置为结果代码。

对于无符号整数，在了解加法结果是否超出数据值的界限方面，进位标志是至关重要的。如果不能确定输入值的长度，在执行无符号整数的加法时，总是应该检查进位标志。如果知道输入值的界限，就可以不必检查进位标志。

处理带符号整数时，进位标志是没有用处的。不仅结果值过大时会设置它，而且只要结果值小于零，也会设置它。虽然它对无符号整数有所帮助，但是对于带符号整数，它是没有意义的（甚至会带来麻烦）。

替换的做法是，在使用带符号整数时，必须关注溢出标志，当结果溢出正值或负值界限时，这个标志会被设置为1。

addtest4.s程序演示在带符号整数加法中使用溢出标志检测错误：

```
# addtest4.s - An example of detecting an overflow condition
.section .data
output:
   .asciz "The result is %d\n"
.section .text
.globl _start
_start:
   movl $-1590876934, %ebx
   movl $-1259230143, %eax
   addl %eax, %ebx
```

```
    jo over
    pushl %ebx
    pushl $output
    call printf
    add  $8, %esp
    pushl $0
    call exit
over:
    pushl $0
    pushl $output
    call printf
    add  $8, %esp
    pushl $0
    call exit
```

addtest4.s程序试图把两个大的负数相加，这造成了溢出情况。JO指令用于检查溢出并且把控制传递到标签over。因为这个程序使用C函数printf，所以要记住把它和系统的动态连接器以及C库连接在一起（关于如何完成这些工作的细节参见第4章）。

如果运行这个程序，它应该生成下面的输出：

```
$ ./addtest4
The result is 0
$
```

输出表明检测到了溢出情况。如果修改MOVL指令，使两个值的相加不产生溢出情况，就会看到加法的结果。例如，下面的值：

```
movl $-190876934, %ebx
movl $-159230143, %eax
```

会生成如下结果：

```
$ ./addtest4
The result is -350107077
$
```

将带符号整数相加时，如果不确定输入数据的长度，那么检查溢出标志以便了解错误情况是很重要的。

3. ADC指令

如果必须处理非常大的、不能存放到双字数据长度（ADD指令可以使用的最大长度）中的带符号或者无符号整数，可以把值分割为多个双字数据元素，并且对每个元素执行独立的加法操作。

为了正确地完成这个工作，必须检测每个加法操作的进位标志。如果进位标志被设置为1，就必须进位到下一对相加的数据元素，如图8-1所示。

如图8-1所示，当最低值的数据元素对相加时，进位标志位必须被进位到下一个值数据元素对，依此类推，直到最高的数据元素对。

手动地完成这一工作，必须使用ADD和JC（或者JO）指令的组合，生成复杂的指令组来确定什么时候产生了进位（或者溢出）情况，并且确定什么时候需要把进位（或者溢出）添加到下一次加法操作中。幸运的是，程序员不必自己处理这些，Intel提供了简单的解决方案。

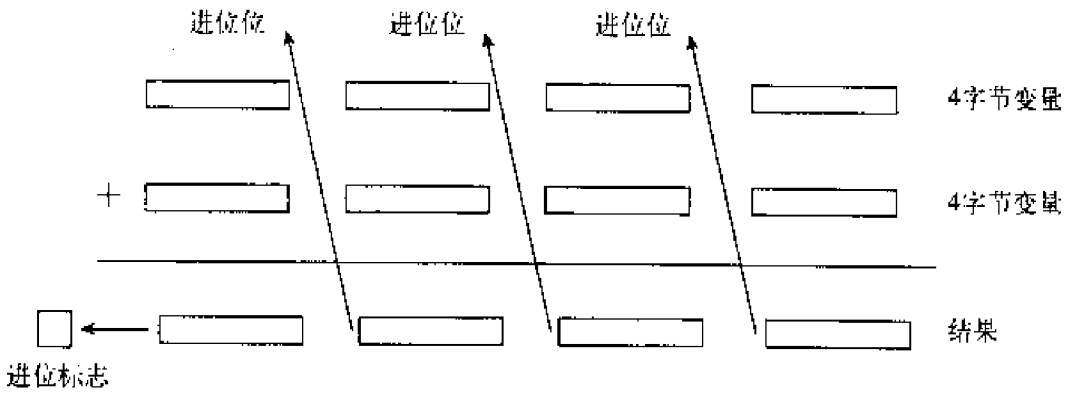



图 8-1

可以使用ADC指令执行两个无符号或者带符号整数值的加法，并且把前一个ADD指令产生的进位标志的值包含在其中。为了执行多组字节的加法操作，可以把多个ADC指令链接在一起，因为ADC指令也按照操作结果正确地设置进位和溢出标志。图8-2显示这种情况。

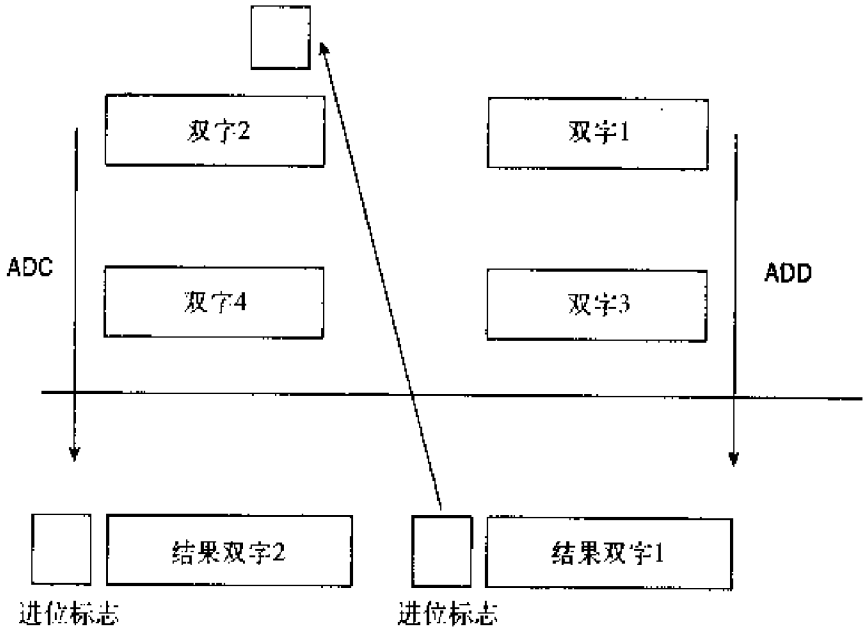



图 8-2

ADC指令的格式如下：

```
adc source, destination
```

其中source可以是立即值或者8位、16位或者32位寄存器或内存位置值，destination可以是8位、16位或者32位寄存器或内存位置值。（和ADD指令类似，source和destination不能同时是内存位置。）还有，和ADD指令一样，GNU汇编器要求在助记符中用附加的字符来表明操作数的长度（b、w或者l）。

4. ADC范例

为了演示如何使用ADC指令，我们编写一个程序把两个大的数字相加：7 252 051 615和5 732 348 928。因为它们都超过了32位无符号整数的界限，所以我们必须使用64位无符号整数值，使用两个32位寄存器保存值。因为使用两个寄存器保存64位值，所以必须通过两次单独的

32位加法把它们相加。低32位相加后，如果进位位被设置为1，就必须把它添加到高32位的加法中。可以使用ADC指令完成这个工作。

一开始，我们需要确定使用哪些寄存器来保存64位值，以及如何使它们相加。图8-3显示这一情况。

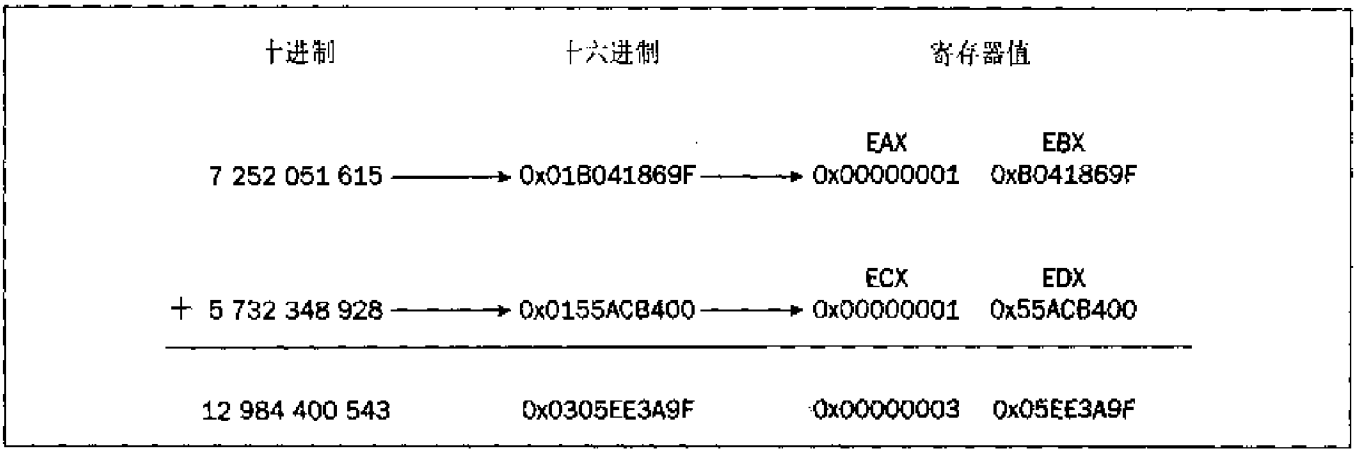



图 8-3

如图8-3所示，EAX: EBX寄存器组合保存第一个值，ECX: EDX寄存器组合保存第二个值。为了把这两个64位值相加，首先，必须使用ADD指令将EBX和EDX寄存器相加。从EBX和EDX寄存器中的值可以看出，加法会生成进位位。之后，EAX和ECX寄存器相加，并且加上第一个加法的进位位。

adctest.s程序演示如何执行这一操作。它把两个64位值相加，一个值保存在EAX: EBX寄存器组合中，另一个值保存在ECX:EDX寄存器组合中：

```
# adctest.s - An example of using the ADC instruction
.section .data
data1:
   .quad 7252051615
data2:
   .quad 5732348928
output:
   .asciz "The result is %qd\n"
.section .text
.globl _start
_start:
   movl data1, %ebx
   movl data1+4, %eax
   movl data2, %edx
   movl data2+4, %ecx
   addl %ebx, %edx
   adcl %eax, %ecx
   pushl %ecx
   pushl %edx
   push $output
   call printf
   addl $12, %esp
   pushl $0
   call exit
```

adctest.s程序首先定义将被相加的两个64位整数值，还有printf函数使用的文本：

```
data1:
   .quad 7252051615
data2:
   .quad 5732348928
output:
   .asciz "The result is %qd\n"
```

printf函数中使用%qd参数来显示64位带符号整数值（如果只使用标准的%d参数，就会只使用32位值）。使用变址寻址方式把64位值加载到EAX:EBX和ECX:EDX寄存器对中：

```
movl data1, %ebx
movl data1+4, %eax
movl data2, %edx
movl data2+4, %ecx
```

data1的值的低32位被加载到EBX寄存器中，高32位被加载到EAX寄存器中。使用相同的方式把data2的值加载到ECX:EDX寄存器对中。

执行前面的操作之后，使用两个指令执行加法操作：

```
addl %ebx, %edx
adcl %eax, %ecx
```

ADDL指令用于两个低位寄存器的加法操作，然后使用ADCL指令执行两个高位寄存器的加法操作，并且加上进位标志。这样确保如果低位寄存器溢出的话，这一情况会被捕捉到并且进位位被添加到高位寄存器中。

执行加法操作之后，64位的结果将保存在ECX: EDX寄存器对中。为了在printf函数中使用它们，必须把它们压入到堆栈中，首先压入包含高位字节的寄存器（ECX）。ECX和EDX对的组合将被C函数printf作为单个64位值读取。

汇编和连接文件之后，可以直接显示输出，或者运行调试器来监视处理过程的各个步骤。例如，在把操作数加载到寄存器中之后，但在执行加法操作之前，寄存器应该如下：

```
(gdb) info reg
eax            0x1      1
ecx            0x1      1
edx            0x55acb400       1437381632
ebx            0xb041869f       -1337882977
```

按照计划，64位整数的十六进制值被加载到了寄存器中。调试器假设寄存器值是带符号整数，所以第三列值是没有意义的。加法指令执行之后，可以再次查看寄存器：

```
(gdb) info reg
eax            0x1      1
ecx            0x3      3
edx            0x5ee3a9f        99498655
ebx            0xb041869f       -1337882977
```

ECX:EDX寄存器对包含结果数据，如图8-3所示。使用printf函数也显示出了十进制形式的结果：

```
$ ./adctest1
The result is 12984400543
$
```

可以随意处理变量data1和data2中的值，改动它们并且在程序执行的过程中查看寄存器是如何加载的。也可以把它们改为负值，这依然产生正确的结果（因为ADD和ADC指令可以处理无符号整数，也可以处理带符号整数）。

### 8.1.2 减法

既然读者已经了解了整数加法操作，学习减法就是轻而易举的事情了。下面几节介绍汇编语言中整数减法的细节。

1. SUB指令

减法的基本形式是SUB指令。和ADD指令一样，SUB指令可以用于无符号和带符号整数。SUB指令的格式如下：

```
sub source, destination
```

其中从destination的值中减去source的值，结果存储在destination操作数的位置。源操作数和目标操作数可以是8位、16位或者32位寄存器或存储在内存中的值（但是再次强调，它们不能同时是内存位置）。源值也可以是立即数值。

和ADD指令一样，GNU汇编器需要在助记符末尾添加长度字符。长度字符仍然是通常使用的字符（b用于字节，w用于字，l用于双字）。

subtest1.s程序演示在汇编语言程序中使用SUB指令：

```
# subtest1.s - An example of the SUB instruction
.section .data
data:
   .int 40
.section .text
.globl _start
_start:
   nop

   movl $0, %eax
   movl $0, %ebx
   movl $0, %ecx
   movb $20, %al
   subb $10, %al
   movsx %al, %eax
   movw $100, %cx
   subw %cx, %bx
   movsx %bx, %ebx
   movl $100, %edx
   subl %eax, %edx
   subl data, %eax
   subl %eax, data
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

subtest1.s程序使用立即值、寄存器和内存位置执行各种基本的减法操作。对程序进行汇编之后，可以在调试器中运行它，并且监视寄存器和内存位置。注意执行SUB指令时值的情况。特别要注意最后一条SUB指令，它把内存位置data1的值（40）减去EAX寄存器中的值（-30）：

```
(gdb) print $eax
$1 = -30
(gdb) x/d &data
0x80490ac <data>:       40
(gdb) s
_start () at subtest1.s:23
23          movl $1, %eax
(gdb) x/d &data
0x80490ac <data>:       70
(gdb)
```

处理器从40中减去-30，并且得到正确的结果70。

记住GNU汇编器的SUB指令中操作数的顺序，这非常重要。使用Intel语法会产生错误的结果！

和SUB指令关系密切的是NEG指令。它生成值的补码。这和使用SUB指令从零中减去这个值是相同的，但是更快。

2. 减法操作中的进位和溢出

和ADD指令类似，执行减法操作之后，SUB指令会修改EFLAGS寄存器的几个位。但是在减法操作中，进位和溢出的概念是不同的。

另外，当加法结果对于保存操作数的数据长度的正界限过大时，会把进位标志设置为1。显然，当减法结果超过数据长度的负界限时，会出现问题。

例如，对于无符号整数，从2中减去5会怎么样？subtest2.s程序演示这个问题：

```
# subtest2.s - An example of a subtraction carry
.section .text
.globl _start
_start:
   nop
   movl $5, %eax
   movl $2, %ebx
   subl %eax, %ebx
   jc under
   movl $1, %eax
   int $0x80
under:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

subtest2.s程序简单地把值5存放到EAX寄存器中，把值2存放到EBX寄存器中，然后从EBX寄存器的值中减去EAX寄存器的值。如果进位标志被设置为1，则使用JC指令进行跳转。程序的结果代码要么是减法的结果值，要么为0（如果进位标志被设置为1）。

对程序进行汇编之后，运行它并且查看情况：

```
$ ./subtest2
$ echo $?
0
$
```

当结果小于零时（这在带符号整数中是非法的），进位标志被设置为1。但是，通过在调试器中检查EBX寄存器的值，应该会发现一些有趣的事情：

```
(gdb) print $ebx
$1 = -3
(gdb)
```

EBX寄存器包含正确的值，尽管它被“认为是”无符号整数。处理器不知道所使用的是无符号整数还是带符号整数。由程序负责确定值是否超出了无符号（或者带符号）值的范围。

使用进位标志确定无符号整数的减法产生负数结果的情况。

和带符号整数加法的情况一样，如果执行带符号整数的减法，进位标志是没有用处的，因为结果常常可能是负值。替换的做法是，必须依靠溢出标志来判断到达了数据长度界限的情况。subtest3.s程序演示这种情况：

```
# subtest3.s - An example of an overflow condition in a SUB instruction
.section .data
output:
   .asciz "The result is %d\n"
.section .text
.globl _start
_start:
   movl $-1590876934, %ebx
   movl $1259230143, %eax
   subl %eax, %ebx
   jo over
   pushl %ebx
   pushl $output
   call printf
   add  $8, %esp
   pushl $0
   call exit
over:
   pushl $0
   pushl $output
   call printf
   add  $8, %esp
   pushl $0
   call exit
```

subtest3.s程序演示从存储在EBX寄存器中的负值中减去存储在EAX寄存器中的正值，生成一个超过32位EBX寄存器范围的值。JO指令用于检测溢出标志，并且把程序转到over: 标签，把输出设置为0。对程序进行汇编并且把程序连接到C库之后，可以运行它并且查看输出：

```
$ ./subtest3
The result is 0
$
```

溢出情况被检测到，并且执行JO指令和进行跳转。可以把赋值给EAX寄存器的值改为负值，进行检测以便确定程序在相反的情况下是否正常工作：

```
movl $-1259230143, %eax
```

再次运行程序：

```
$ ./subtest3
The result is -331646791
$
```

这一次，减去负数生成一个绝对值更小的负数，它在数据长度的界限之内，没有设置溢出标志。

3. SBB指令

和加法操作一样，可以使用进位情况帮助执行大的无符号整数值的减法操作。SBB指令在多字节减法操作中利用进位和溢出标志实现跨越数据边界的借位特性。

SBB指令的格式如下:

```
sbb source, destination
```

其中进位位被添加到source值，然后从destination值中减去source值得到结果。结果存储在destination位置中。照例，源和目标值可以是8位、16位或者32位寄存器或内存中的值，当然，不能同时使用内存位置作为源和目标值。

SBB指令最常用于从前一条SUB指令“挖出”进位标志。当前一条SUB指令被执行并且造成进位时，进位位被SBB指令“借走”以便继续进行下一个数据对的减法。图8-4演示这一情况。

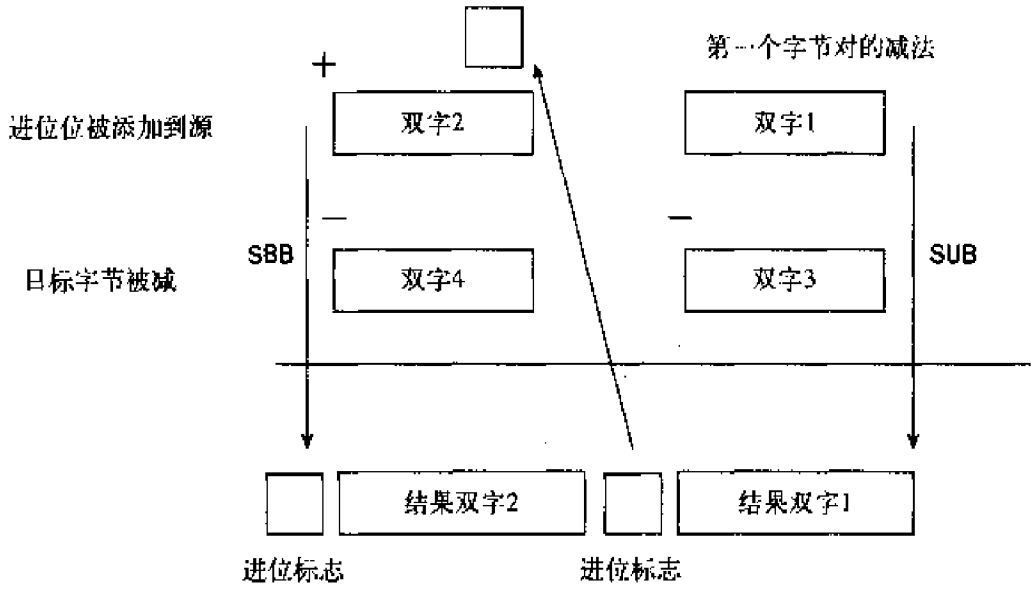



图 8-4

下面的sbbtest.s程序演示在多字节减法操作中使用SBB指令:

```
# sbbtest.s - An example of using the SBB instruction
.section .data
data1:
   .quad 7252051615
data2:
   .quad 5732348928
output:
   .asciz "The result is %qd\n"
.section .text
.globl _start
_start:
   nop
   movl data1, %ebx
   movl data1+4, %eax
   movl data2, %edx
   movl data2+4, %ecx
```

```
subl %ebx, %edx
sbbl %eax, %ecx
pushl %ecx
pushl %edx
push $output
call printf
add  $12, %esp
pushl $0
call exit
```

读者也许会注意到sbbtest.s程序和adctest.s程序完全一样，除了使用SUB和SBB指令替换了ADD和ADC指令之外。这个程序定义了两个四字值，把它们加载到寄存器中，然后使用SUB/SBB指令组合进行减法操作。

对程序进行汇编，并且把它和C库进行连接之后，应该得到下面的结果:

```
$ ./sbbtest
The result is -1519702687
$
```

可以改动四字data1和data2的值，以便确定这个程序可以处理合法的64位值的任何组合情况。

### 8.1.3 递增和递减

当创建汇编语言程序时，常常必须遍历数据数组以便处理每个元素。这种情况在程序设计领域中太常见了，以至于Intel提供了专门的指令来自动地提供计数功能。

INC和DEC指令用于对无符号整数值进行递增（INC）和递减（DEC）操作。INC和DEC指令不会影响进位标志，所以可以递增或者递减计数器的值，并且不会影响程序循环中涉及进位标志的任何其他加法或者减法操作。

这两个指令的格式如下

```
dec destination
inc destination
```

其中destination可以是8位、16位或者32位寄存器，或者内存中的值。

记住INC和DEC指令主要用于无符号整数。如果对设置为0的32位寄存器进行递减，新的值将是0xFFFFFFFF，它看上去就像带符号整数-1，但是它被当作无符号整数4 294 967 295处理（没有设置正确的标志）。如果把它们用于带符号整数，就要小心符号的变化。

### 8.1.4 乘法

在整数运算中比较复杂的功能之一是乘法。与加法和减法不同，对于无符号和带符号整数的乘法操作，需要不同的指令。本节介绍用于整数乘法的指令，以及如何在程序中使用它们。

1. 使用MUL进行无符号整数乘法

MUL指令用于两个无符号整数相乘。它的格式可能和预期的有所不同。MUL指令的格式如下:

```
mul source
```

其中source可以是8位、16位或者32位寄存器或内存值。读者也许会奇怪在这个指令行中只

提供一个操作数，怎么能进行两个值的乘法。答案是目标操作数是隐含的。

使用隐含目标操作数的情况有些复杂。首先，目标位置总是使用EAX寄存器的某种形式，这取决于源操作数的长度。因此，根据源操作数的值的长度，乘法操作中使用的另一个操作数必须存放在AL、AX或者EAX寄存器中。

由于乘法可能产生很大的值，所以MUL指令的目标位置必须是源操作数的两倍长度。如果源值是8位，那么目标操作数就是AX寄存器，因为结果是16位。当源操作数更大时，情况甚至会更加复杂。

不幸的是，当和16位源操作数相乘时，EAX寄存器不被用于保存32位结果。为了向下兼容老式的处理器，Intel使用DX：AX寄存器对保存32位乘法结果值（这一格式源自16位处理器的年代）。结果的高位字存储在DX寄存器中，低位字存储在AX寄存器中。

对于32位源值，目标位置使用64位EDX：EAX寄存器对，高位双字存储在EDX寄存器中，低位双字在EAX寄存器中。当使用MUL的16位或者32位版本时，如果在EDX（或者DX）寄存器中存储着数据，那么一定要把数据保存到其他位置。

为了帮助总结这些情况，下表列出无符号整数乘法的需求。

| 源操作数长度 | 目标操作数 | 目标位置 |
|---|---|---|
| 8位 | AL | AX |
| 16位 | AX | DX:AX |
| 32位 | EAX | EDX:AX |

另外，记住这一点很重要：使用GNU汇编器时，必须在助记符的结尾加上正确的长度字符。

2. MUL指令范例

作为使用MUL指令的一个例子，我们把315 814和165 432相乘。图8-5演示MUL指令如何处理这两个值。

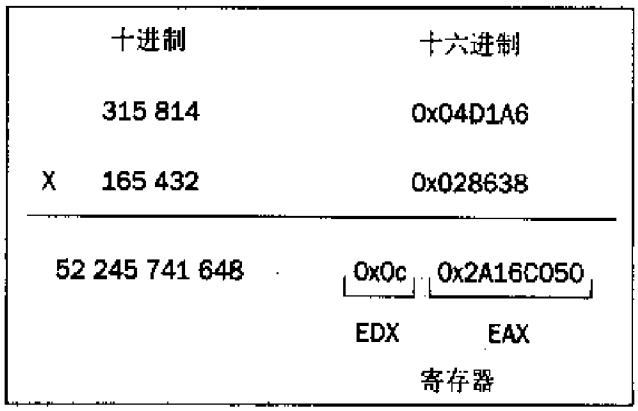



图 8-5

MUL指令中使用的两个操作数都是32位值。乘法的结果被存储为64位值，分割存放在EDX和EAX寄存器中。

multest.s程序演示两个32位无符号整数的乘法操作，并且从EDX:EAX寄存器获得结果：

```
# multest.s - An example of using the MUL instruction
.section .data
data1:
   .int 315814
data2:
   .int 165432
result:
   .quad 0
output:
   .asciz "The result is %qd\n"
.section .text
.globl _start
```

```
_start:
    nop
    movl data1, %eax
    mull data2
    movl %eax, result
    movl %edx, result+4
    pushl %edx
    pushl %eax
    pushl $output
    call printf
    add $12, %esp
    pushl $0
    call exit
```

multest.s程序在内存中定义两个整数值（记住使用MUL指令时，它们都必须是无符号值），把其中一个值加载到EAX寄存器中，然后使用MUL指令把另一个值和EAX寄存器中的值相乘。从EDX:EAX寄存器对得到的结果被加载到一个64位内存位置中（使用变址内存访问方式），并且使用C函数printf显示结果。

对程序进行汇编，并且把它和C库连接之后，可以直接运行它查看输出，还可以在调试器中运行它以便查看执行过程中寄存器的情况。MUL指令执行之后，调试器的输出应该如下：

```
(gdb) print/x $eax
$3 = 0x2a16c050
(gdb) print/x $edx
$4 = 0xc
(gdb) x/gd &result
0x80491c4 <result>:     52245741648
(gdb) x/8b &result
0x80491c4 <result>: 0x50  0xc0  0x16  0x2a  0x0c  0x00 0x00 0x00
(gdb)
```

EDX: EAX寄存器对组合生成结果值，这个值被存储在内存位置result中，并且通过printf函数显示出来。

3. 使用IMUL进行带符号整数乘法

MUL指令只能用于无符号整数，而IMUL指令可以用于带符号和无符号整数，但是必须小心结果不使用目标的最高有效位。对于较大的值，IMUL指令只对带符号整数是合法的。为了应付比较复杂的情况，IMUL指令有3种不同的指令格式。

IMUL指令的第一种格式使用一个操作数，其行为和MUL指令完全一样：

```
imul source
```

source操作数可以是8位、16位或者32位寄存器或内存中的值，它与位于AL、AX或者EAX寄存器（取决于源操作数的长度）中的隐含操作数相乘。然后，结果被存放到AX寄存器、DX:AX寄存器对或者EDX:EAX寄存器对中。

IMUL指令的第二种格式允许指定EAX寄存器之外的目标操作数：

```
imul source, destination
```

其中source可以是16位或者32位寄存器或内存中的值，destination必须是16位或者32位通用寄存器。这种格式允许指定把乘法操作的结果存放到哪个位置（而不是强制使用AX和DX寄存器）。

这种格式的缺陷在于乘法操作的结果被限制为单一目标寄存器的长度（非64位结果）。使用这种格式时必须非常小心，不要溢出目标寄存器（在乘法操作之后，使用第6章“控制执行流程”中介绍的标准方法检查进位或者溢出标志，以便确保结果适合目标寄存器）。

IMUL指令的第三种格式允许指定3个操作数：

```
imul multiplier, source, destination
```

其中multiplier是一个立即值，source是16位或者32位寄存器或内存中的值，destination必须是通用寄存器。这种格式允许执行一个值（source）和一个带符号整数（multiplier）的快速乘法操作，把结果存储到通用寄存器（destination）中。

和MUL指令一样，在GNU汇编器中使用IMUL助记符，要记住在IMUL指令的结尾添加长度字符，以便指定源和目标操作数的长度。

4. IMUL指令范例

熟悉MUL指令之后，使用IMUL指令的第一种格式就是小事情了。其他两种格式需要一些例子来演示，所以下面的imultest.s程序演示IMUL指令的后两种格式：

```
# imultest.s - An example of the IMUL instruction formats
.section .data
value1:
   .int 10
value2:
   .int -35
value3:
   .int 400
.section .text
.globl _start
_start:
   nop
   movl value1, %ebx
   movl value2, %ecx
   imull %ebx, %ecx
   movl value3, %edx
   imull $2, %edx, %eax
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

imultest.s程序创建要处理的一些带符号整数（value1、value2和value3），把它们传送到寄存器中，然后使用IMUL指令的不同格式执行一些乘法操作。

汇编和连接程序之后，可以使用调试器监视程序执行过程中寄存器的值。单步执行IMUL指令之后，寄存器应该如下：

```
(gdb) info reg
eax            0x320    800
ecx            0xfffffea2       -350
edx            0x190    400
ebx            0xa      10
```

EAX寄存器包含EDX寄存器的值（400）和立即值2相乘得到的结果。ECX寄存器包含EBX寄存器的值（10）和最初加载到ECX寄存器中的值（-35）相乘的结果。注意，结果作为带符号

整数值被存放到ECX寄存器中。

5. 检查溢出

当使用带符号整数和IMUL指令时，记住这一点很重要：总是要检查结果中的溢出。完成这个工作的一种方式是使用JO指令检查溢出标志（另一种方式是检查进位标志）。

imultest2.s程序演示这种做法：

```
# imultest2.s - An example of detecting an IMUL overflow
.section .text
.globl _start
_start:
   nop
   movw $680, %ax
   movw $100, %cx
   imulw %cx
   jo over
   movl $1, %eax
   movl $0, %ebx
   int $0x80
over:
   movl $1, %eax
   movl $1, %ebx
   int $0x80
```

imultest2.s程序把两个值传送到16位寄存器中（AX和CX），然后使用16位IMUL指令将它们相乘。设置结果会导致16位寄存器溢出，并且JO指令跳转到标签over，这里退出程序，带有结果代码1。如果修改加载到寄存器中的立即数值，使结果小于65 535，IMUL指令就不会把溢出标志设置为1，不会执行JO指令，程序退出，带有结果代码0。

### 8.1.5 除法

和乘法类似，根据使用的是无符号还是带符号整数，除法操作分别需要特定的指令。整数除法的难点是答案不总是精确的整数，比如你用9除以2。这会生成两部分答案。商是被除数中包含除数的个数。余数是剩余了多少（答案的小数部分）。作为除法的结果，除法指令生成商和余数两个部分。

本节介绍并且演示用于整数除法的DIV和IDIV指令。

1. 无符号除法

DIV指令用于无符号整数的除法操作。DIV指令的格式如下：

```
div divisor
```

其中divisor（除数）是隐含的被除数要除以的值，它可以是8位、16位或者32位寄存器或内存中的值。在执行DIV指令之前，被除数必须已经存储到了AX寄存器（对于16位值）、DX:AX寄存器对（对于32位值）或者EDX:EAX寄存器对（对于64位值）。

允许的除数的最大值取决于被除数的长度。对于16位被除数，除数只能是8位；对于32位被除数，除数只能是16位；对于64位被除数，除数只能是32位。

除法操作的结果是两个单独的数字：商和余数。这两个值都存储在被除数值使用的相同寄存器中。下表列出了其设置的情况。

| 被除数 | 被除数长度 | 商 | 余数 |
|---|---|---|---|
| AX | 16位 | AL | AH |
| DX:AX | 32位 | AX | DX |
| EDX:EAX | 64位 | EAX | EDX |

这就是说，当除法操作完成时，会丢失被除数，所以要确保这不是这个值的唯一拷贝（除非在除法操作之后就不需要被除数的值了）。还要记住，结果会改变DX或者EDX寄存器的值，所以也要小心其中存储的内容。

divtest.s程序演示简单的除法操作：

```
# divtest.s - An example of the DIV instruction
.section .data
dividend:
   .quad 8335
divisor:
   .int 25
quotient:
   .int 0
remainder:
   .int 0
output:
   .asciz "The quotient is %d, and the remainder is %d\n"
.section .text
.globl _start
_start:
   nop
   movl dividend, %eax
   movl dividend+4, %edx
   divl divisor
   movl %eax, quotient
   movl %edx, remainder
   pushl remainder
   pushl quotient
   pushl $output
   call printf
   add  $12, %esp
   pushl $0
   call exit
```

divtest.s程序把一个64位四字整数加载到EDX:EAX寄存器对中（记住内存中的小尾数顺序和寄存器中的大尾数顺序是相反的，这在第5章“传送数据”中讨论过），然后使用一个存储在内存中的32位双字整数除这个值。32位的商值存储在一个内存位置中，32位的余数值存储在另一个内存位置中。

对程序进行汇编并且把它和C库连接之后，可以使用不同的被除数和除数值运行它（记住这些值都必须是正值，并且小于它们的数据长度允许的最大值）。可以使用调试器监视寄存器中的值是如何被处理的。

2. 带符号除法

IDIV指令的使用方式和DIV指令完全一样，但是它用于带符号整数的除法操作。它也使用

隐含的被除数，被除数位于AX寄存器、DX:AX寄存器对或者EDX:EAX寄存器对中。

和IMUL指令不同，IDIV指令只有一种格式，它指定除法操作中使用的除数：

```
idiv divisor
```

同样，其中的divisor可以是8位、16位或者32位寄存器或内存中的值。

IDIV指令把结果返回和DIV指令相同的寄存器中，并且商和余数的格式也是相同的（除了结果是带符号整数之外）。

对于带符号整数的除法，余数的符号总是与被除数的符号相同。

关于带符号除法，另一件要记住的事情是被除数的长度。因为它必须是除数长度的两倍，所以有时候必须把整数值扩展为适当的数据长度（参见第7章）。使用符号扩展指令（比如MOVSX）把除法操作的被除数扩展为适当的数据长度是很重要的。扩展操作的失败将导致错误的被除数值，并且在结果中产生错误。

3. 检查除法错误

整数除法最大的问题在于检查产生错误条件的情况，比如发生除以零的情况，或者商（或余数）溢出目标寄存器。

发生错误时，系统会产生中断，这会在Linux系统中产生一个错误，比如下面这样：

```
$ ./divtest
Floating point exception
$
```

产生这个错误是由于在divtest.s程序中把divisor的值设置成了0。

在程序中执行DIV和IDIV指令之前，检查除数和被除数的值是程序员的责任。不进行这样的检查可能导致应用程序中出现错误行为。

## 8.2 移位指令

乘法和除法是处理器上最为耗费时间的两种指令。但是，可以运用一些技巧帮助加快应用程序的执行速度。移位指令提供了执行基于2的乘方的乘法和除法操作的快速和容易的方式。本节介绍使用移位指令执行乘法和除法操作的方法。

移位利用了二进制运算的一种特性，使乘以和除以2的乘方的操作更加简单。读者也许熟悉十进制领域，并且熟悉如何把十进制值移位到另一个十进制位，使这个值自动地乘以或者除以10的乘方（例如，把值2移动一个十进制位到20的操作和使它乘以10是相同的）。

同样的原则也适用于二进制数字，只不过使用的是2的乘方。把二进制数字向左移1位就是使之乘以2，移2位就是使之乘以4，3位就是使之乘以8，等等。数据元素中的移位操作比执行二进制乘法操作要快得多，可以使用这种方法提高数学运算密集型程序的性能。

下面几节介绍如何在程序中执行移位指令。

### 8.2.1 移位乘法

为了使整数乘以2的乘方，必须把值向左移位。可以使用两个指令使整数值向左移位——SAL（向左算术移位）和SHL（向左逻辑移位）。这两个指令执行相同的操作，并且是可以互换

的。它们有3种不同格式:

```
sal destination
sal %cl, destination
sal shifter, destination
```

第一种格式把destination的值向左移1位，这等同于使值乘以2。

第二种格式把destination的值向左移动CL寄存器中指定的位数。

最后一个版本把destination的值向左移动shifter值指定的位数。在所有的格式中，目标操作数可以是8位、16位或者32位寄存器或内存中的值。

和以往一样，GNU汇编器需要在助记符的结尾附加上一个字符，用于指出目标值的长度。

可以对带符号和无符号整数执行向左移位指令。移位造成的空位用零填充。移位造成的超出数据长度的任何位首先被存放在进位标志中，然后在下一次移位操作中被丢弃。因此，如果值的最高有效位为1，经过2次向左移位操作之后，最高有效位就会从进位标志中丢弃。图8-6演示这种情况。

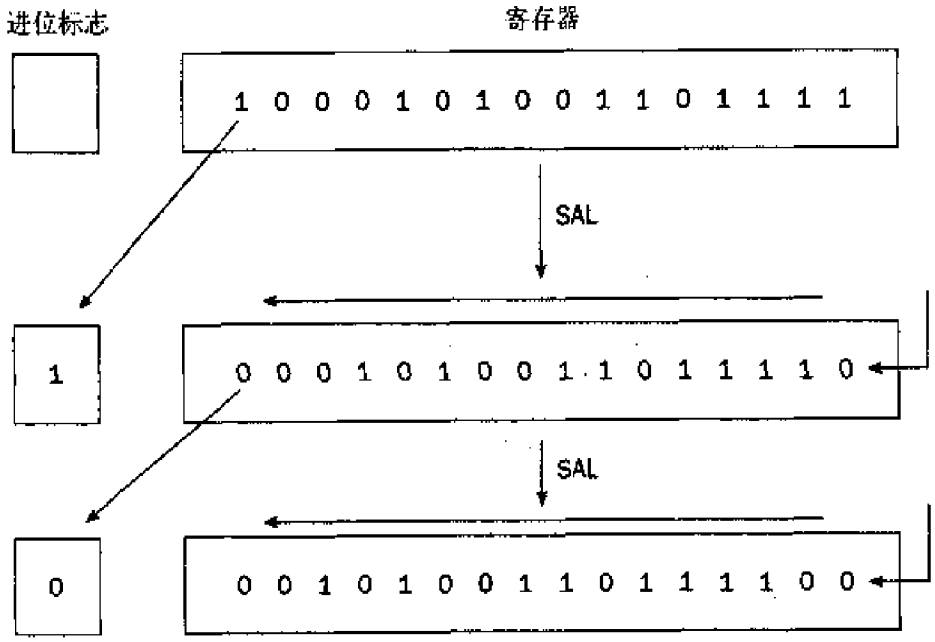



图 8-6

saltest.s程序演示使用SAL指令的基本情况:

```
# saltest.s - An example of the SAL instruction
.section .data
value1:
   .int 25
.section .text
.globl _start
_start:
   nop
   movl $10, %ebx
   sall %ebx
   movb $2, %cl
   sall %cl, %ebx
   sall $2, %ebx
   sall value1
```

```
sall $2, value1
movl $1, %eax
movl $0, %ebx
int $0x80
```

saltest.s程序演示SAL指令的全部3种格式，使用EBX寄存器保存要移位的值。对程序进行汇编之后，可以使用调试器监视寄存器的值：

```
(gdb) info reg
eax            0x0      0
ecx            0x2      2
edx            0x0      0
ebx            0x140    320
(gdb) x/d &value1
0x804909c <value1>:      200
(gdb)
```

一开始，十进制值10被加载到EBX寄存器中。第一条SAL指令把它移动1位（使之乘以2，结果为20）。第二条SAL指令把它移动2位（使之乘以4，结果为80），第三条SAL指令把它再移动2位（使之乘以4，结果为320）。value1位置中的值（25）被移动1位（使之为50），然后再移动2位（使之为200）。

### 8.2.2 移位除法

通过移位进行除法操作涉及把二进制值向右移位。但是，当把整数值向右移位时，必须要注意整数的符号。

对于无符号整数，向右移位产生的空位可以被填充为零，而且不会有任何问题。不幸的是，对于带符号整数，使用零填充高位部分会对负数产生有害的影响。

为了解决这个问题，有两个向右移位指令。SHR指令清空移位造成的空位，所以它只能用于对无符号整数进行移位操作。SAR指令根据整数的符号位，要么清空，要么设置移位造成的空位。对于负数，空位被设置为1，但是对于正数，它们被清空为0。

和向左移位指令一样，向右移位指令把位移出数据元素。移出数据元素的任何位（最低有效位）首先被移动到进位标志，然后移出去（丢弃）。图8-7演示这种情况。

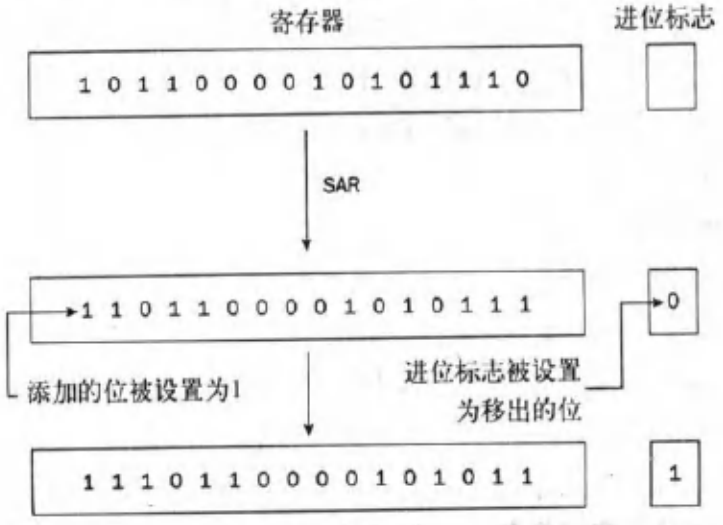



图 8-7

### 8.2.3 循环移位

和移位指令关系密切的指令是循环移位指令。循环移位指令执行的功能和移位指令一样，只不过溢出位被存放回值的另一端，而不是被丢弃。例如，字节值的向左循环移位操作获得第7位中的值，并且把它存放到第0位的位置，其他每个位的位置向左移动1位。

下表列出可以使用的各种循环移位指令。

| 指　令 | 描　述 |
|---|---|
| ROL | 向左循环移位 |
| ROR | 向右循环移位 |
| RCL | 向左循环移位，并且包含进位标志 |
| RCR | 向右循环移位，并且包含进位标志 |

最后两条指令使用进位标志作为附加位的位置，来支持9位移位。循环移位指令的格式和移位指令相同，提供3种选择：

- 单一操作数：按照指定的方向把它移动1位。
- 2个操作数：指定循环次数的%cl寄存器和目标操作数。
- 2个操作数：指定循环次数的立即值和目标操作数。

## 8.3 十进制运算

第7章中介绍过，二进制编码的十进制（Binary Coded Decimal，BCD）格式是用于处理人可读的数字的常见方法，在处理器中可以快速地处理这种格式。虽然很多高级BCD处理操作位于FPU中，但是核心处理器包含一些简化的指令，用于使用BCD值执行运算。

本节介绍可用的基本BCD运算指令，并且演示如何在汇编语言程序中使用它们。

### 8.3.1 不打包BCD的运算

在第7章中介绍过，不打包的BCD值在一个字节中包含单个十进制位（0到9）。多个十进制位存储在多个字节中，每个十进制位占用1个字节。当应用程序需要对不打包的BCD值执行数学操作时，应用程序假设结果也应该按照不打包BCD格式存储。幸运的是，IA-32平台提供了专门的指令用于从一般数学操作生成不打包BCD值。

用于把二进制运算结果转换为不打包BCD格式的指令有4条：

- AAA：调整加法操作的结果。
- AAS：调整减法操作的结果。
- AAM：调整乘法操作的结果。
- AAD：准备除法操作的被除数。

这些指令必须和一般的无符号整数指令ADD、ADC、SUB、SBB、MUL和DIV组合在一起使用。AAA、AAS和AAM指令在它们各自的操作之后使用，把二进制结果转换为不打包BCD格式。AAD指令有些不同，在DIV指令之前使用它，用于准备被除数以便生成不打包BCD结果。

这些指令都使用一个隐含的操作数——AL寄存器。AAA、AAS和AAM指令假设前一个操作

的结果存放在AL寄存器中，并且把这个值转换为不打包BCD格式。AAD指令假设被除数以不打包BCD格式存放在AX寄存器中，并且把它转换为DIV指令要处理的二进制格式。结果是正确的一个不打包BCD值、AL寄存器中的商和AH寄存器中的余数（按照不打包BCD格式）。

当处理多字节的不打包BCD值时，必须使用进位和溢出标志才能确保计算出正确的值。图8-8演示这个问题。

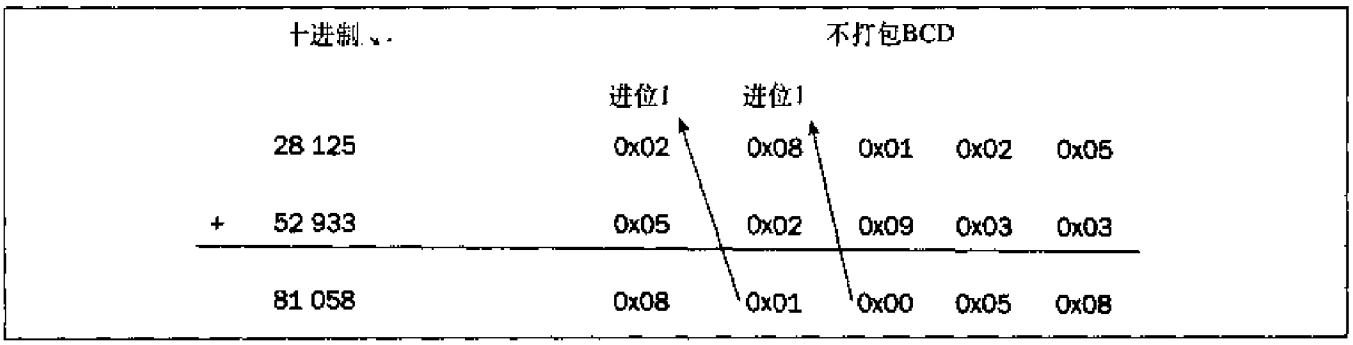



图 8-8

对第三组不打包BCD值执行的加法操作生成了进位，必须把它带到下一个BCD值中才能生成正确的结果。AAA、AAS和AAM指令都把AH寄存器和进位标志一起使用来表明何时需要进位操作。

最好在程序中演示这种情况。aaatest.s程序演示两个多字节不打包BCD值的加法操作:

```
# aaatest.s - An example of using the AAA instruction
.section .data
value1:
   .byte 0x05, 0x02, 0x01, 0x08, 0x02
value2:
   .byte 0x03, 0x03, 0x09, 0x02, 0x05
.section .bss
   .lcomm sum, 6
.section .text
.globl _start
_start:
   nop
   xor %edi, %edi
   movl $5, %ecx
   clc
loop1:
   movb value1(, %edi, 1), %al
   adcb value2(, %edi, 1), %al
   aaa
   movb %al, sum(, %edi, 1)
   inc %edi
   loop loop1
   adcb $0, sum(, %edi, 4)
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

不打包BCD值在内存位置中存储为小尾数格式。第一个值被一次1个字节地读取到AL寄存器中，并且使用ADC指令把它和第二个值中相同位置的值相加。使用ADC指令是为了确保加上前一

次加法操作产生的任何进位位（不要忘记在循环之前使用CLC指令确保清空了进位标志）。在ADC指令之后使用AAA指令把AL寄存器中的二进制结果转换为不打包BCD格式，然后把结果存储在位于sum位置中的相同位置。ECX寄存器对值的位置进行计数，以便LOOP指令知道何时退出。

对程序进行汇编之后，可以通过在调试器中运行它并且监视每个处理步骤的过程中的执行情况来检测程序。例如，第三次执行ADC指令之后（第三个值位置），AL寄存器包含以下值：

```
(gdb) info reg
eax            0xa      10
(gdb)
```

它显示9和1的二进制加法结果为10。但是，执行AAA指令之后，AX寄存器的值如下：

```
(gdb) info reg
eax            0x100    256
(gdb)
```

它显示AH寄存器中的不打包值为1，AL寄存器中为0。1被带入到下一位的值的加法操作。最后，结果按照不打包BCD格式存放到内存位置sum中：

```
(gdb) x/6b &sum
0x80490b8 <sum>:        0x08    0x05    0x00    0x01    0x08    0x00
(gdb)
```

因此，28 125和52 933的相加结果为81 058。

### 8.3.2 打包BCD的运算

处理打包BCD值时，可用的指令只有2条：

- DAA：调整ADD或者ADC指令的结果。
- DAS：调整SUB或者SBB指令的结果。

这些指令执行的功能和AAA以及AAS指令相同，但是操作对象是打包BCD值。它们也使用位于AL寄存器中的隐含操作数，并且把转换结果存放在AL寄存器中，把进位位存放在AH寄存器和辅助进位标志位中。

图8-9显示打包BCD运算的一个例子。

| | 十进制 | 打包BCD | 以小尾数格式的打包BCD |
|---|---|---|---|
| | 52 933 | 0x05 0x29 0x33 | 0x33 0x29 0x05 |
| — | 28 125 | 0x02 0x81 0x25 | 0x25 0x81 0x02 |
| | 24 808 | 0x02 0x48 0x08 | 0x08 0x48 0x02 |

图 8-9

打包BCD值52 933按照小尾数格式（0x332905）加载到内存中，并从它减去BCD值28 125（0x258102）。打包BCD格式的结果为0x084802。

dastest.s程序演示使用SBB和DAS指令执行这个减法操作:

```
# dastest.s - An example of using the DAS instruction
.section .data
value1:
   .byte 0x25, 0x81, 0x02
value2:
   .byte 0x33, 0x29, 0x05
.section .bss
   .lcomm result, 4
.section .text
.globl _start
_start:
   nop
   xor %edi, %edi
   movl $3, %ecx
loop1:
   movb value2(, %edi, 1), %al
   sbbb value1(, %edi, 1), %al
   das
   movb %al, result(, %edi, 1)
   inc %edi
   loop loop1
   sbbb $0, result(, %edi, 4)
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

dastest.s程序把第一个打包BCD值加载到AL寄存器中（每次一个十进制位）。然后使用SBB指令从它减去第二个打包BCD值。这样，前一次减法操作留下的任何进位位都会被考虑在内。然后使用DAS指令把结果转换为将存储在内存位置result中的打包BCD格式。ECX寄存器用于控制必须经过的循环次数（每个打包BCD字节一次）。转换之后，如果留有剩下的进位位，就把它存放在结果值中。

汇编和连接程序之后，可以在调试器中运行它，并且随着通过SBB指令计算减法值，然后通过DAS指令把它转变为打包BCD格式，监视EAX寄存器的值。例如，第一个减法操作之后，EAX寄存器的值如下:

```
(gdb) info reg
eax            0x0e       14
(gdb)
```

但是执行DAS指令之后，这个值改变为:

```
(gdb) info reg
eax            0x08       8
(gdb)
```

它表示结果的第一个十进制位。

## 8.4 逻辑操作

除了标准的加法、减法、乘法和除法运算功能之外，汇编语言还提供了对字节值中包含的

原始位执行各种操作的指令。本节介绍汇编语言程序员经常使用的两种常见的位功能：布尔逻辑和位测试。

### 8.4.1 布尔逻辑

处理二进制数字时，具有可用的标准布尔逻辑功能是很方便的。提供的布尔逻辑操作如下：

- AND
- NOT
- OR
- XOR

AND、OR和XOR指令使用相同的格式：

```
and source, destination
```

其中source可以是8位、16位或者32位立即值、寄存器或内存中的值，destination可以是8位、16位或者32位寄存器或内存中的值（但是和以往一样，不能同时使用内存值作为源和目标）。NOT指令使用单一操作数，它既是源值，也是目标结果的位置。

布尔逻辑功能对源和目标执行按位操作。就是说，使用指定的逻辑功能，按照顺序对数据元素的每个位进行单独比较。读者已经看到过在汇编语言程序中使用布尔逻辑指令的一些例子。有几个范例程序使用XOR指令用零填充寄存器。清空寄存器的最高效的方式是使用OR指令对寄存器和它本身进行异或操作。当和本身进行XOR操作时，每个设置为1的位就变为0，每个设置为0的位也变为0。这确保寄存器的所有位都被设置为0，这比使用MOV指令加载立即值0的方式要快。

### 8.4.2 位测试

有时候必须确定值内的单一位是否被设置为1了。这一功能最常见的用途是检查EFLAGS寄存器标志的值。不需要试图比较整个寄存器的值，好的方式是检测单一标志的值。

完成这个工作的一种方式是使用AND指令，把EFLAGS寄存器和已知位值进行比较，挑选出希望检查的一个位或者多个位。但是，程序员也许不希望改变包含EFLAGS位的寄存器的值。

为了解决这个问题，IA-32平台提供了TEST指令。TEST指令在8位、16位或32位值之间执行按位逻辑AND操作，并且相应地设置符号、零和奇偶校验标志，而且不修改目标值。

TEST指令的格式和AND指令相同。尽管没有数据写入目标位置，但是仍然必须指定任意立即值作为源值。这类似于CMP指令的工作方式和SUB指令一样，但是它不会把结果存储到任何位置。

刚才提到过，TEST指令最常见的用途是检查EFLAGS寄存器中的标志。例如，如果希望使用CPUID指令检查处理器的属性，程序员首先应该确保处理器支持CPUID指令。

EFLAGS寄存器中的ID标志（第21位）用于确定处理器是否支持CPUID指令。如果可以修改ID标志，则说明CPUID指令是可用的。为了进行测试，必须获得EFLAGS寄存器，反转ID标志位，然后再测试这个位，查看它是否真的改变了。cpuidtest.s程序执行这些操作：

```
# cpuidtest.s - An example of using the TEST instruction
.section .data
output_cpuid:
   .asciz "This processor supports the CPUID instruction\n"
output_nocpuid:
   .asciz "This processor does not support the CPUID instruction\n"
.section .text
.globl _start
_start:
   nop
   pushfl
   popl %eax
   movl %eax, %edx
   xor $0x00200000, %eax
   pushl %eax
   popfl
   pushfl
   popl %eax
   xor %edx, %eax
   test $0x00200000, %eax
   jnz cpuid
   pushl $output_nocpuid
   call printf
   add  $4, %esp
   pushl $0
   call exit
cpuid:
   pushl $output_cpuid
   call printf
   add  $4, %esp
   pushl $0
   call exit
```

cpuidtest.s程序首先使用PUSHFL指令把EFLAGS寄存器的值保存到堆栈顶部。然后，使用POPL指令把EFLAGS值读取到EAX寄存器中。

下一个步骤演示如何使用XOR指令设置寄存器的一位。使用MOVL指令把EFLAGS值的拷贝保存到EDX寄存器中，然后使用XOR指令设置ID位（仍然在EAX寄存器中）为值1。XOR指令使用一个设置了ID位的立即值。EAX寄存器经过异或操作之后，就确保ID位被设置为1了。下一个步骤把新的EAX寄存器值压入到堆栈中，然后使用POPFL指令把它存储在EFLAGS寄存器中。

现在必须确定是否成功地设置了ID标志。再一次使用PUSHFL指令把EFLAGS寄存器压入堆栈，然后使用POPL指令把它弹出到EAX寄存器中。这个值和原始的EFLAGS值（先前存储在EDX寄存器中）进行XOR操作，查看值改变成了什么。

最后，使用TEST指令查看ID标志位是否改变了。如果是，那么EAX中的值就不为零，然后使用JNZ指令进行跳转，输出适当的消息。

## 8.5 小结

本章介绍了很多背景知识。我们讨论了基本的整数运算，介绍了如何对无符号和带符号整数执行基本的加法、减法、乘法和除法操作。

ADD和ADC指令用于无符号和带符号整数的加法操作，ADC指令用于包含多字节值的进位标

志。SUB和SBB用于无符号和带符号整数的减法操作，SBB指令用于包含多字节值的进位标志。

接下来，讨论了乘法和除法。不幸的是，对于乘法和除法，无符号和带符号整数需要单独的指令。MUL和DIV用于无符号整数的乘法和除法，IMUL和IDIV指令用于带符号整数的乘法和除法。本章还介绍了移位指令，这是乘以和除以2的乘方的快速方式。SAL和SHL指令执行值的向左算术移位或者向左逻辑移位，这是值的乘法操作的快捷方式。SAR和SHR执行执行值的向右算术移位或者向右逻辑移位，这是值的除法操作的快捷方式。

十进制运算可以在数学功能中使用打包和不打包的BCD值。和二进制模式相比，BCD值提供了更加容易阅读的显示整数值的方法。AAA、AAS、AAM和AAD指令提供执行不打包BCD值的加法、减法、乘法和除法操作的方法。它们必须和二进制运算指令结合在一起使用。打包BCD功能包括DAA和DAS，它们用于打包BCD值的加法和减法操作。

最后，本章讨论如何使用AND、OR、NOT和XOR指令实现常见的布尔逻辑功能。这些指令可以对寄存器和内存中的二进制值执行按位布尔操作。TEST指令提供执行AND指令的简单方式，而且不修改目标值。这是测试二进制值（比如EFLAGS寄存器内的标志值）的理想方式。

下一章研究FPU领域。FPU提供高级数学运算功能，用于处理浮点运算和其他的BCD运算。如果程序员必须在工程技术或者科学研究的环境下进行程序设计，这些功能就是至关重要的。

# 第9章 高级数学功能

在早期，浮点运算要么使用软件模拟，要么需要单独的运算协处理器。从80486开始，Intel在板载的FPU中整合了浮点操作（参见第2章）。本章介绍FPU内包含的浮点操作并演示如何在IA-32平台上实现浮点运算。

本章的第一部分介绍FPU的布局，扼要回顾第7章中介绍过的用于把数字加载到FPU中和从FPU获取结果的指令。然后，介绍基本浮点运算功能：加法、减法、乘法和除法。之后，将学习如何使用更加高级的浮点运算功能，比如平方根和三角函数。接下来，介绍用于比较浮点数的方法，然后介绍在内存中存储FPU环境以及从备份恢复FPU环境的方法。

## 9.1 FPU环境

第2章描述过IA-32平台上FPU环境的基础知识。既然读者已经比较熟悉IA-32平台的布局和操作了，现在可以深入研究FPU基础结构以及用于控制它的指令了。本节描述FPU寄存器堆栈；控制字，它们控制FPU如何操作；状态字，它们表明FPU中发生了什么，以及标记字，它们定义FPU寄存器堆栈中包含的值。

### 9.1.1 FPU寄存器堆栈

第2章曾经提过，FPU是一个自持的单元，它使用与标准处理器寄存器分离的一组寄存器处理浮点操作。附加的FPU寄存器包括8个80位数据寄存器和3个16位寄存器，称为控制（control）、状态（status）和标记（tag）寄存器。

FPU数据寄存器称为R0到R7（但是读者将会看到，不使用这些名称访问它们）。它们的操作和标准寄存器有些不同，不同之处在于它们连接在一起形成一个堆栈。和内存中的堆栈不同，FPU寄存器堆栈是循环的——这就是说，堆栈中的最后一个寄存器连接回堆栈中的第一个寄存器。

堆栈顶部的寄存器是在FPU的控制字寄存器中定义的，名为ST(0)。除了顶部寄存器外的其他寄存器名称是ST(x)，其中x可以是1到7。如图9-1所示。

当数据被加载到FPU堆栈时，堆栈顶部沿着8个寄存器向下移动。当8个值被加载到堆栈中之后，所有8个FPU数据寄存器就都被使用了。如果把第9个数据加载到堆栈中，堆栈指针回绕到第一个寄存器，并且使用新的值替换这个寄存器中的值，这会产生FPU异常错误。

第7章介绍了如何使用FLD指令把浮点值存放到FPU堆栈中，如何使用FILD指令把整数存放到FPU堆栈中，如何使用FBLD指令把BCD数据存放到FPU堆栈中。各种浮点常量值也可以用于把常量值加载到堆栈中。还有用于把FPU寄存器中的值按照每种不同的数据类型存储到内存位置中的命令。

### 9.1.2 FPU状态、控制和标记寄存器

因为FPU独立于主处理器，所以它一般不使用EFLAGS寄存器来表示结果和确定行为。FPU

包含它自己的寄存器组来执行这些功能。状态寄存器、控制寄存器和标记寄存器用于存取FPU的特性和确定FPU的状态。

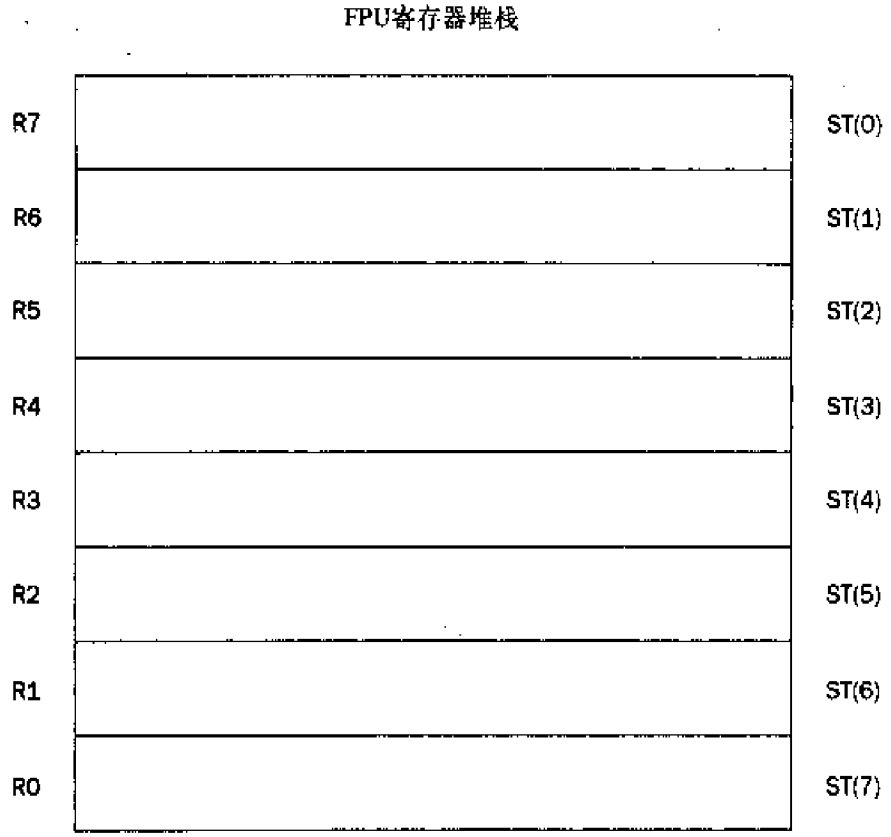



图 9-1

本节介绍这3个FPU寄存器，以及如何在程序中访问它们。

1. 状态寄存器

状态寄存器表明FPU的操作情况。它包含在一个16位寄存器中，不同的位作为不同标志。下表介绍状态寄存器位。

| 状态位 | 描　述 |
|---|---|
| 0 | 非法操作异常标志 |
| 1 | 非规格化操作数异常标志 |
| 2 | 除数为零异常标志 |
| 3 | 溢出异常标志 |
| 4 | 下溢异常标志 |
| 5 | 精度异常标志 |
| 6 | 堆栈错误 |
| 7 | 错误汇总状态 |
| 8 | 条件代码位0（C0） |
| 9 | 条件代码位1（C1） |
| 10 | 条件代码位2（C2） |
| 11-13 | 堆栈顶部指针 |
| 14 | 条件代码位3（C3） |
| 15 | FPU繁忙标志 |

4个条件代码位（8、9、10和14位）一起使用，表示浮点操作结果的特定错误代码。它们经常和异常标志一起使用，表示特定的异常情况。在本章后面，读者会更多地看到这些位的使用。

FPU的前6位是异常标志。在处理的过程中，当发生浮点异常时FPU设置它们。FPU保持它们的设置状态，直到程序运行时清空它们。当检测到堆栈溢出或者下溢时（值对于80位堆栈寄存器过大或者过小），设置堆栈错误标志。

堆栈顶部位用于表示哪个FPU数据寄存器被设置为ST0寄存器。8个寄存器中的任何一个都可以被指派为堆栈的顶端。每个后续的寄存器被相应地设为ST(x)寄存器。

把值加载到堆栈中时，在值被加载之前，TOS值递减1。这样，因为默认的TOS值为0，所以R7寄存器是堆栈顶部值的默认位置（ST0）。这可能引起混乱，但是不必担心——FPU堆栈会处理所有这些管理事务。

使用FSTSW指令，可以把状态寄存器读取到一个双字内存位置或者AX寄存器中。getstatus.s程序演示这种情况：

```
# getstatus.s - Get the FPU Status register contents
.section .bss
   .lcomm status, 2
.section .text
.globl _start
_start:
   nop
   fstsw %ax
   fstsw status

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

汇编和连接程序之后，可以在调试器中运行它，以查看存放在AX寄存器和内存位置status中的值：

```
(gdb) x/x &status
0x804908c <status>:     0x00000000
(gdb) print/x $eax
$1 = 0x0
(gdb)
```

这两个位置产生相同的值，说明在默认情况下FPU状态寄存器的所有位都被设置为零。也可以使用info all命令在调试器中查看FPU的状态、控制和标记寄存器的值：

```
(gdb) info all
.
.
.
fctrl          0x37f    895
fstat          0x0      0
ftag           0x55555  349525
(gdb)
```

上面显示了这3个寄存器的当前值。

2. 控制寄存器

控制寄存器控制FPU内的浮点功能。这里定义了相关设置，比如FPU用于计算浮点值的精度以及用于舍入浮点结果的方法。

控制寄存器使用一个16位寄存器，下表列出了相应位的含义。

| 控制位 | 描　述 |
|---|---|
| 0 | 非法操作异常掩码 |
| 1 | 非规格化操作数异常掩码 |
| 2 | 除数为零异常掩码 |
| 3 | 溢出异常掩码 |
| 4 | 下溢异常掩码 |
| 5 | 精度异常掩码 |
| 6-7 | 保留 |
| 8-9 | 精度控制 |
| 10-11 | 舍入控制 |
| 12 | 无穷大控制 |
| 13-15 | 保留 |

控制寄存器的前6位用于控制使用状态寄存器中的哪些异常标志。当这些位中的一位被设置时，就会防止状态寄存器中对应的异常标志被设置。默认情况下，所有掩码位都被设置，即屏蔽所有异常。

精度控制位可以设置FPU中用于数学计算的浮点精度。这是非常有用的控制特性，可以改变FPU计算浮点值花费的时间。精度控制位可能的设置如下:

- 00——单精度（24位有效位）
- 01——未使用
- 10——双精度（53位有效位）
- 11——扩展双精度（64位有效位）

默认情况下，FPU精度被设置为扩展双精度。这是最为精确的值，但是也最耗费时间。如果不打算使用这么高的精度，可以把这个值设置为单精度以便加快浮点值的计算速度。

类似地，舍入控制位可以设置FPU如何舍入浮点计算的结果。舍入控制位的可能设置如下:

- 00——舍入到最近值
- 01——向下舍入（向无穷大负值）
- 10——向上舍入（向无穷大正值）
- 11——向零舍入

默认情况下，舍入控制位被设置为舍入到最近值。

控制寄存器的默认值是0x037F。可以使用FSTCW指令把控制寄存器的设置加载到双字内存位置中查看设置的内容。也可以使用FLDCW指令改变设置。这条指令把双字内存值加载到控制寄存器中。setprec.s程序使用FLDCW指令把FPU的精度设置从扩展双精度改为单精度:

```
# setprec.s - An example of setting the precision bits in the Control Register
.section .data
newvalue:
```

```
    .byte 0x7f, 0x00
.section .bss
    .lcomm control, 2
.section .text
.globl _start
_start:
    nop
    fstcw control
    fldcw newvalue
    fstcw control

    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

setprec.s程序把双字值newvalue定义为0x07f（在内存中存储字节时，要记住使用小尾数格式）。这个值把精度位设置为00，即设置FPU精度为单精度浮点。然后程序使用FSTCW指令获得当前控制寄存器的设置，存放到双字内存位置control中，然后使用FLDCW指令把newvalue的值加载到控制寄存器中。为了确保这个值被正确地存储了，再次使用FSTCW指令检查当前控制寄存器的值。

汇编和连接程序之后，可以在调试器中单步执行指令并且监视控制寄存器的值：

```
(gdb) run
Starting program: /home/rich/palp/chap09/setprec

Breakpoint 1, _start () at setprec.s:11
11          fstcw control
(gdb) x/x &control
0x804909c <control>:    0x00000000
(gdb) s
12          fldcw newvalue
(gdb) x/x &control
0x804909c <control>:    0x0000037f
(gdb) s
13          fstcw control
(gdb) s
15          movl $1, %eax
(gdb) x/x &control
0x804909c <control>:    0x0000007f
(gdb) info all
.
.
fctrl          0x7f     127
(gdb)
```

控制寄存器成功地被设置成了0x07f，所以现在FPU使用单精度浮点计算。

这样做并不一定会加快所有浮点计算的速度。能够显示出改进的最常见的功能是除法和平方根计算。

3. 标记寄存器

标记寄存器用于标识8个80位FPU数据寄存器中的值。标记寄存器使用16位（每个寄存器2位）标识每个FPU数据寄存器的内容，如图9-2所示。

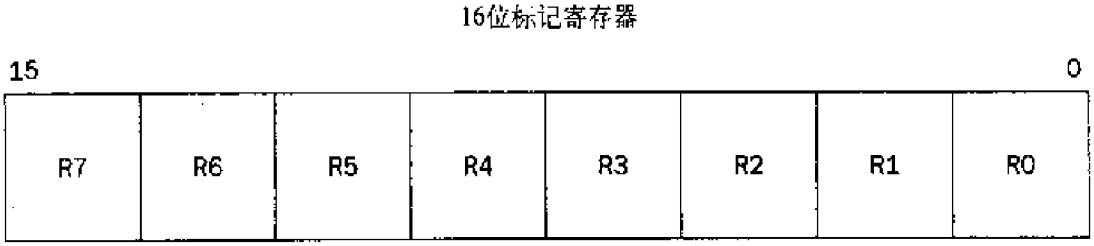



图 9-2

每个标记值对应一个物理的FPU寄存器。每个寄存器对应的2位值可以包含表明寄存器内容的4个特殊代码之一。在任何给定的时刻，FPU数据寄存器可以包含下面的内容：

- 一个合法的扩展双精度值（代码00）
- 零值（代码01）
- 特殊的浮点值（代码10）
- 无内容（空）（代码11）

这使程序员可以快速检查标记寄存器以便确定FPU寄存器中是否包含合法数据，而不必读取和分析寄存器的内容，虽然在实际操作中，因为是程序员把值压入到寄存器堆栈中的，所以程序员应该知道其中的内容是什么。

### 9.1.3 使用FPU堆栈

第7章简要介绍过把浮点值加载到FPU寄存器堆栈中。为了执行浮点运算，了解如何在FPU堆栈中操作数据是至关重要的。所有FPU数学操作都是在这里执行的。程序员必须了解如何把数据调度到堆栈中以及如何在堆栈中调度数据以便执行计算。下面的stacktest.s程序演示如何把各种数据类型加载到FPU堆栈中，还有处理FPU堆栈时使用的一些常见的堆栈功能：

```
# stacktest.s - An example of working with the FPU stack
.section .data
value1:
   .int 40
value2:
   .float 92.4405
value3:
   .double 221.440321
.section .bss
   .lcomm int1, 4
   .lcomm control, 2
   .lcomm status, 2
   .lcomm result, 4
.section .text
.globl _start
_start:
   nop
   finit
   fstcw control
   fstsw status
   fildl value1
   fistl int1
   fldl value2
```

```
fldl value3
fst %st(4)
fxch %st(1)
fstps result
movl $1, %eax
movl $0, %ebx
int $0x80
```

这个简单的程序中进行了很多操作，所以我们慢慢研究。首先使用FINIT指令初始化FPU。它把控制和状态寄存器设置为它们的默认值，但是不改变FPU数据寄存器中包含的数据。最好在使用FPU的任何程序中都包含这个指令。

接下来，使用FSTCW和FSTSW指令把FPU控制和状态寄存器复制到内存位置。指令执行之后，可以通过查看这些内存位置的值来查看这些寄存器的默认值：

```
(gdb) x/2b &control
0x80490cc <control>:    0x7f    0x03
(gdb) x/2b &status
0x80490ce <status>:     0x00    0x00
(gdb)
```

输出显示控制寄存器的默认值是0x037f（记住存放在内存中的值的格式是小尾数格式），状态寄存器的默认值是0x0000。

下一条指令（FILDS）把一个双字整数值加载到FPU寄存器堆栈中。FISTS指令获取寄存器堆栈顶部的值（刚刚存放到那里的值），并且把它存放到目标位置（它被设置为内存位置int1）：

```
(gdb) info all
.
.
st0            40        (raw 0x4004a000000000000000)
(gdb) x/d &int1
0x80490c8 <int1>:       40
(gdb)
```

整数值40存储到被标记为堆栈顶部的寄存器（表示为ST0）。但是，注意存储的值的十六进制值。非常容易发现它没有被存储为一般的带符号整数值。而是当值被存储到FPU寄存器中时，它被转换为扩展双精度浮点数据类型。从FPU寄存器堆栈获取值并且把它存放到内存中的时候，它被自动转换回双字整数（因为FIST助记符指定了S字符）。可以通过查看这个内存位置的十六进制值进行检查：

```
(gdb) x/4b &int1
0x80490c8 <int1>:       0x28    0x00    0x00    0x00
(gdb)
```

正如我们期望的，值在内存中被存储为双字带符号整数值。

下面两条指令把浮点值加载到FPU寄存器堆栈中。第一条使用FLDS指令加载内存位置value2中的单精度浮点值。第二条使用FLDL指令加载内存位置value3中的双精度浮点值。现在有3个值被加载到了FPU寄存器堆栈中。每个值被加载的时候，相对于堆栈的顶部，前面的值向堆栈的下方移动。

FLD指令执行之后，FPU寄存器堆栈应该如下：

```
(gdb) info all
.
.
st0    221.44032100000001150874595623463392  (raw 0x4006dd70b8e086bdf800)
st1    92.44049835205078125  (raw 0x4005b8e1890000000000)
st2    40    (raw 0x4004a000000000000000)
(gdb)
```

当使用info all命令显示FPU寄存器时，读者也许注意到其他FPU数据寄存器或许包含，或许不包含无关数据。当执行FINIT指令时，它不会初始化FPU数据寄存器，但是会改变标记值，显示它们是空的。所以可能有其他操作遗留下的无关数据。程序员要跟踪程序使用哪些FPU数据寄存器，以及哪些寄存器中包含合法数据。

最后，3条FPU指令在寄存器之间进行一些数据传送操作。FST指令用于把ST0寄存器的数据传送到另一个FPU寄存器。注意用于指定从堆栈顶部开始的第5个FPU寄存器的格式。GNU汇编器使用百分号表示寄存器值，而且FPU寄存器的引用号必须用括号括起来。

FST指令执行之后，使用FXCH指令交换ST0寄存器和另一个FPU寄存器的值——在这个例子中，另一个寄存器是ST1。这两条指令执行之后，FPU寄存器应该如下：

```
(gdb) info all
.
.
st0    92.44049835205078125  (raw 0x4005b8e1890000000000)
st1    221.44032100000001150874595623463392  (raw 0x4006dd70b8e086bdf800)
st2    40    (raw 0x4004a000000000000000)
st3    0     (raw 0x00000000000000000000)
st4    221.44032100000001150874595623463392  (raw 0x4006dd70b8e086bdf800)
(gdb)
```

围绕FPU寄存器堆栈调度数据并且执行了必须的数学操作之后，很可能需要从FPU寄存器堆栈获取结果。FST和FSTP指令也可以用于把数据从FPU寄存器传送到内存位置。FST指令把数据从FPU寄存器ST0复制到内存位置（或者另一个FPU寄存器），同时保持ST0寄存器中的原始值不变。

FSTP指令也复制FPU寄存器ST0中的值，但是之后把值从FPU寄存器堆栈弹出。这把FPU堆栈中所有堆栈值向上移动一个位置。

不要忘记在FST和FSTP助记符的结尾添加数据长度字符以便指定结果数据值的正确长度。在这个例子中，使用FSTPS指令从FPU堆栈的ST0位置中的值创建一个存储在4个字节（32位）内存中的单精度浮点值：

```
(gdb) x/f &result
0x80490cc <result>:    92.4404984
(gdb) x/4b &result
0x80490cc <result>:    0x89    0xe1    0xb8    0x42
(gdb)
```

FSTPS指令执行之后，可以看到值从堆栈中删除了，并且其他值向上“移动”了一个位置：

```
(gdb) info all
.
.
```

```
st0   221.44032100000001150874595623463392     (raw 0x4006dd70b8e086800)
st1   40        (raw 0x4004a000000000000000)
st2   0         (raw 0x00000000000000000000)
st3   221.44032100000001150874595623463392     (raw 0x4006dd70b8e086800)
(gdb)
```

既然读者已经了解了如何在FPU中操作浮点值，现在该开始对数据执行数学操作了。

## 9.2 基本浮点运算

正如我们所料，FPU提供对浮点值执行基本数学功能的指令。下表介绍这些基本功能。

| 指 令 | 描 述 |
|---|---|
| FADD | 浮点加法 |
| FDIV | 浮点除法 |
| FDIVR | 反向浮点除法 |
| FMUL | 浮点乘法 |
| FSUB | 浮点减法 |
| FSUBR | 反向浮点减法 |

实际上，这些功能的每一个都具有单独的指令和格式，可以生成6个可能的功能，这取决于希望执行的确切操作是什么。例如，FADD指令可以像下面这样使用：

- FADD source：内存中的32位或者64位值和ST0寄存器相加
- FADD %st(x)，%st(0)：st(x)和st(0)相加，结果存储到st(0)中
- FADD %st(0)，%st(x)：st(0)和st(x)相加，结果存储到st(x)中
- FADDP %st(0)，%st(x)：st(0)和st(x)相加，结果存储到st(x)中，并且弹出st(0)
- FADDP：st(0)和st(1)相加，结果存储到st(1)中，并且弹出st(0)
- FIADD source：16位或者32位整数值和st(0)相加，结果存储到st(0)中

每种不同的格式指定操作中使用哪个FPU寄存器，以及操作完成后如何处理寄存器（要么保留，要么弹出堆栈）。跟踪FPU寄存器值的状态很重要。有时候，复杂的数学操作会执行多个操作，这些操作把各种值存储到不同的寄存器中，这时跟踪FPU寄存器值的状态可能是困难的工作。

对于GNU汇编器，情况甚至变得更加复杂。指定内存中的值的指令的助记符必须包含一个字符的长度指示符（s用于32位单精度浮点值，l用于双精度浮点值）。而且，像以往一样，源和目标操作数的顺序和Intel语法中的顺序是相反的。

下面是使用浮点运算指令的一些例子：

```
fadds data1          # add the 32-bit value at data1 to the ST0 register
fmull data1          # multiply the 64-bit value at data1 with the ST0 register
fidiv data1         # divide ST0 by the 32-bit integer value at data1
fsub %st, %st(1)     # subtract the value in ST0 from ST1, and store in ST1
fsub %st(0), %st(1) # subtract the value in ST0 from ST1, and store in ST1
fsub %st(1), %st(0) #subtract the value in ST1 from ST0, and store in ST0
```

FSUBR和FDIVR指令用于执行反向减法和除法——就是说，运算结果是目标值减去（或者除）源值，并且把结果存放在目标操作数位置。这与FSUB和FDIV指令执行运算的方式是相反（反向）的。当希望交换数学表达式中的顺序时，这些指令很方便，不必使用额外的指令在FPU

寄存器之间传送数据。

为了演示这些指令如何工作，我们使用IA-32 FPU指令完成一个复杂的数学操作。这个数学操作计算如下表达式:

```
((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) - (140.2 / 94.21))
```

为了解决这个问题，最好确定如何在FPU寄存器中存储和移动值。在执行数学操作时，把尽可能多的值加载到FPU中总可以提高速度，而且不必在FPU寄存器和内存之间来回交换值。新的值被加载时，原始值在FPU寄存器堆栈中“向下移动”。跟踪如何在堆栈中安排值是很重要的。

首先，必须考虑如何把值加载到FPU堆栈中，以及操作将如何影响答案位置的安排。下面是对应该怎样执行计算的逐步分析:

1) 把43.65加载到ST0中。

2) ST0除以22，结果保存在ST0中。

3) 把76.34加载到ST0中（步骤2的答案移动到ST1）。

4) 把3.1加载到ST0中（步骤3的值移动到ST1，步骤2的答案移动到ST2）。

5) ST0和ST1相乘，答案存放在ST0中。

6) ST0和ST2相加，答案存放在ST0中（这是方程式的左半边的答案）。

7) 把12.43加载到ST0中（步骤6的答案移动到ST1）。

8) ST0乘以6，答案存放在ST0中。

9) 把140.2加载到ST0中（步骤8的答案移动到ST1，步骤6的答案移动到ST2）。

10) 把94.21加载到ST0中（步骤8的答案移动到ST2，步骤6的答案移动到ST3）。

11) ST1除以ST0，弹出堆栈并且把结果保存到ST0中（步骤8的答案移动到ST1，步骤6的答案移动到ST2）。

12) ST1减去ST0，结果存储在ST0中（这是方程式的右半边的答案）。

13) ST2除以ST0，结果存储在ST0中（这是最终的答案）。

试图判断各个值位于FPU寄存器堆栈的什么位置时常常容易使人困惑。有时将处理的进展情况形象化会有所帮助。图9-3演示前面的计算次序。

各个步骤被绘制到堆栈的图表上之后，就非常容易发现值存储在什么位置。既然已经准备就绪，现在可以开始编写代码来实现它了。fpmath1.s程序使用FPU指令实现这个表达式计算:

```
# fpmath1.s - An example of basic FPU math
.section .data
value1:
   .float 43.65
value2:
   .int 22
value3:
   .float 76.34
value4:
   .float 3.1
value5:
   .float 12.43
value6:
   .int 6
value7:
   .float 140.2
```

```
value8:
   .float 94.21
output:
   .asciz "The result is %f\n"
.section .text
.globl _start
_start:
nop
   finit
   flds value1
   fidiv value2
   flds value3
   flds value4
   fmul %st(1), %st(0)
   fadd %st(2), %st(0)
   flds value5
   fimul value6
   flds value7
   flds value8
   fdivrp
   fsubr %st(1), %st(0)
   fdivr %st(2), %st(0)
   subl  $8, %esp
   fstpl (%esp)
   pushl $output
   call printf
   add  $12, %esp
   pushl $0
   call exit
```

fpmath1.s程序在数据段中定义计算中将用到的所有变量。然后进行计算，把一些值加载到FPU寄存器中，并且在可能的时候使用一些值作为指令的操作数。有一件要注意的特殊事情，当目标地址（ST0）是除数（或者要减去的数）的值时，使用FDIVR和FSUBR指令很方便。

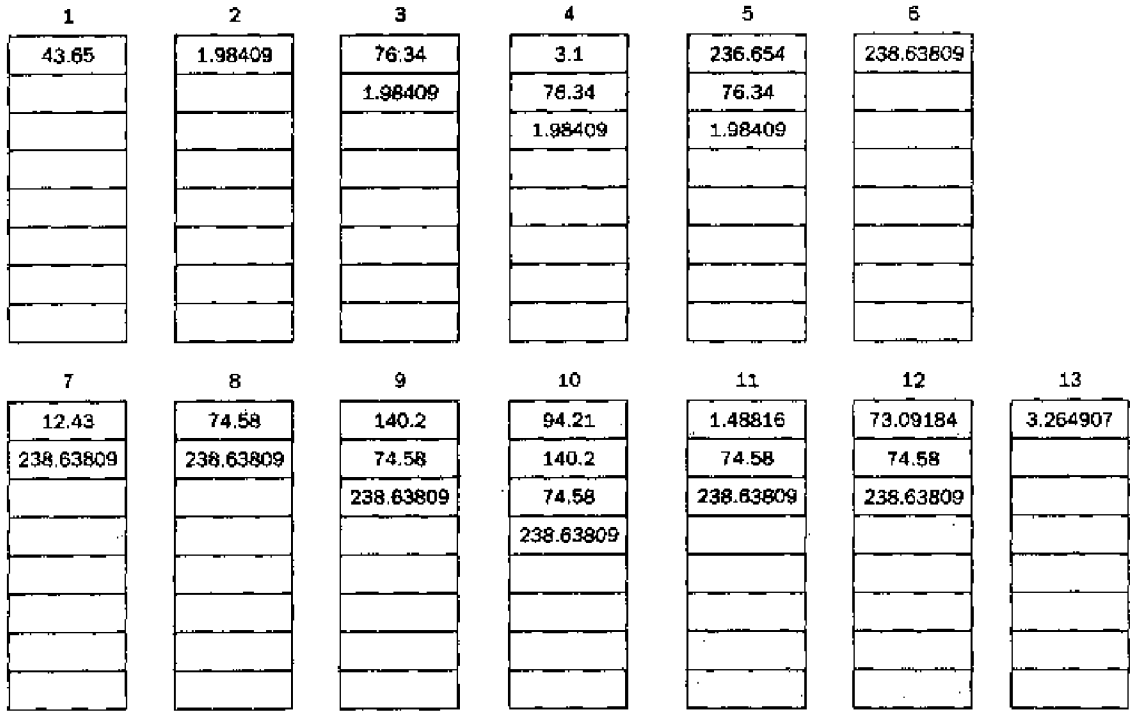



图 9-3

计算完成之后，答案被放在FPU寄存器ST0中。使用FSTPL指令将这个值弹出FPU寄存器堆栈，并且在这个例子中，使用ESP寄存器值将它放在程序堆栈的顶部（先要通过将ESP减8在堆栈上保留8字节）。这确保C函数printf能够使用这个值。printf函数需要双精度格式的浮点值，所以必须使用FSTPL指令。

对程序进行汇编并且把它连接到C库之后，可以从命令行运行它以便查看是否会得到正确的答案:

```
$ ./fpmath1
The result is 3.264907
$
```

生成的答案和我得到的答案（当然，使用计算器）相同。如果非常关心幕后的情况如何，可以在调试器中运行程序并且监视指令如何控制FPU寄存器。

## 9.3 高级浮点运算

除了简单的加法、减法、乘法和除法之外，还有许多其他的浮点运算。FPU提供很多对浮点数执行的高级功能。如果为科学研究或者工程技术应用编写汇编语言程序，就很可能必须在程序中引入高级运算功能。

下表介绍可用的高级功能。

| 指　令 | 描　述 |
|---|---|
| F2XM1 | 计算2的乘方（次数为ST0中的值）减去1 |
| FABS | 计算ST0中的值的绝对值 |
| FCHS | 改变ST0中的值的符号 |
| FCOS | 计算ST0中的值的余弦 |
| FPATAN | 计算ST0中的值的部分反正切 |
| FPREM | 计算ST0中的值除以ST1中的值的部分余数 |
| FPREM1 | 计算ST0中的值除以ST1中的值的IEEE部分余数 |
| FPTAN | 计算ST0中的值的部分正切 |
| FRNDINT | 把ST0中的值舍入到最近的整数 |
| FSCALE | 计算ST0乘以2的ST1次乘方 |
| FSIN | 计算ST0中的值的正弦 |
| FSINCOS | 计算ST0中的值的正弦和余弦 |
| FSQRT | 计算ST0中的值的平方根 |
| FYL2X | 计算ST1*log ST0（以2为基数） |
| FYL2XP1 | 计算ST1*log（ST0+1）（以2为基数） |

前面的大多数功能的含义不言自明。下面几节比较详细地介绍其中一些功能。

### 9.3.1 浮点功能

FABS、FCHS、FRNDINT和FSQRT指令对浮点值执行简单的数学功能。FABS指令计算ST(0)的绝对值。FCHS改变值的符号。FSQRT计算ST(0)的平方根。

fpmath2.s程序演示这些指令的使用:

```
# fpmath2.s - An example of the FABS, FCHS, and FSQRT instructions
.section .data
value1:
   .float 395.21
value2:
   .float -9145.290
value3:
   .float 64.0
.section .text
.globl _start
_start:
   nop
   finit
   flds value1
   fchs
   flds value2
   fabs
   flds value3
   fsqrt

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

汇编和连接程序之后，可以在调试器中监视FPU寄存器。在指令执行结束之后，FPU寄存器应该如下：

```
(gdb) info all
.
.
.
st0        8          (raw 0x40028000000000000000)
st1        9145.2900390625  (raw 0x400c8ee5290000000000)
st2        -395.209991455078125      (raw 0xc007c59ae10000000000)
(gdb)
```

记住，值的顺序与它们被压入到FPU堆栈中时的顺序相反。ST0寄存器保存FSQRT指令的结果，ST2保存FCHS指令的结果。

FRNDINT指令的不同之处在于它的行为取决于FPU控制寄存器中的舍入位的值。按照前面小节中介绍的4种舍入方法之一，FRNDINT指令把ST0中的浮点值舍入到最近的整数值。roundtest.s程序演示这种情况：

```
# roundtest.s - An example of the FRNDINT instruction
.section .data
value1:
   .float 3.65
rdown:
   .byte 0x7f, 0x07
rup:
   .byte 0x7f, 0x0b
.section .bss
   .lcomm result1, 4
   .lcomm result2, 4
   .lcomm result3, 4
.section .text
```

```
.globl _start
_start:
   nop
   finit
   flds value1
   frndint
   fists result1

   fldcw rdown
   flds value1
   frndint
   fists result2

   fldcw rup
   flds value1
   frndint
   fists result3

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

roundtest.s程序定义两个双字值（rdown和rup），使用它们改变FPU控制寄存器中的舍入位。因为其他值不需要改变，我们可以使用一个静态值完成这个工作。为了把舍入方向设置为向下，舍入位被设置为二进制值01，这使控制寄存器的值为0x77F。为了把舍入方向设置为向上，舍入位被设置为二进制值10，这使控制寄存器的值为0xB7F。

第一组指令初始化FPU，把测试值加载到ST0中，执行FRNDINT指令（使用默认的舍入设置），然后把结果传送到内存位置result1中（作为整数值）：

```
finit
flds value1
frndint
fists result1
```

下一组指令把设置舍入方向为向下的值加载到控制寄存器中，把测试值加载到ST0寄存器中，执行舍入功能，然后把结果存储到内存位置result2中：

```
fldcw rdown
flds value1
frndint
fists result2
```

最后一组指令把设置舍入方向为向上的值加载到控制寄存器中，把测试值加载到ST0中，执行舍入功能，然后把结果存储到内存位置result3中。

汇编和连接程序之后，可以在调试器中运行程序并且监视它如何工作。第一组指令执行之后，舍入后的值应该在内存位置result1中：

```
(gdb) x/d &result1
0x80490c4 <result1>:    4
(gdb)
```

默认情况下，浮点值会向上舍入到整数值4。下一组指令执行之后，舍入后的值应该在内存位置result2中：

```
(gdb) x/d &result2
0x80490c8 <result2>:    3
(gdb)
```

正如我们期望的，通过把舍入位设置为向下，新的舍入后的值被设置为整数值3。最后，执行完最后一组指令之后，舍入后的值应该在内存位置result3中：

```
(gdb) x/d &result3
0x80490cc <result3>:    4
(gdb)
```

结果显示现在的舍入位使值向上舍入到最近的整数。

### 9.3.2 部分余数

部分余数是浮点除法中难于处理的一个部分。部分余数的概念与如何执行浮点除法有关。除法操作的余数由被除数对除数的一系列减法决定。在每次减法迭代时，中间余数称为部分余数（partial remainder）。当部分余数小于除数时（不能再执行减法操作，否则就会产生负数），迭代停止。在除法操作结束时，最终的答案是代表减法迭代次数的整数值（称为商（quotient）），以及表示最终的部分余数的浮点值（现在称为余数（remainder））。

根据执行除法需要多少次迭代，可能有很多部分余数。迭代的次数取决于被除数和除数的指数值之间的差值。每次减法不能使被除数的指数值的减少量超过63。

FPREM和FPREM1指令都计算浮点除法的余数值，但是它们的工作方法稍有区别。

确定除法余数的基本方法是确定被除数和除数的除法的浮点商，然后把这个值舍入到最近的整数。那么，余数就是除数和商相乘的结果与被除数之间的差值。例如，为了计算20.65除以3.97的余数，可以执行如下步骤：

1) 20.65/3.97 = 5.201511335，舍入到5（这是商）

2) 5*3.97 = 19.85

3) 20.65−19.85 = 0.8（这是余数）

困难的部分在于舍入过程。在创建部分余数的任何标准之前，Intel就开发了FPREM指令。Intel的开发人员选择使用默认的FPU向零舍入的方法，用于计算整数商值，然后确定余数。

不幸的是，当IEEE创建标准时，它选择在计算余数之前，使商值向上舍入到最近的整数值。虽然这似乎只有细微的区别，但是在处理过程中计算部分余数时造成很大影响。出于这个原因，Intel选择保持原始FPREM指令的原始形式，并且另外创建了FPREM1指令，它使用IEEE方法计算部分余数。

计算部分余数的问题在于必须知道迭代过程在什么时候完成。FPREM和FPREM1指令都使用FPU状态寄存器的条件代码位2（状态寄存器的第10位）表示迭代何时完成。当需要更多的迭代时，就设置C2位，当迭代完成时，就清空C2位。

为了检查C2位，必须首先使用FSTSW指令把状态寄存器的内容复制到内存位置或者AX寄存器中，然后使用TEST指令判断这一位是否设置了。

premtest.s程序使用FPREM1指令执行简单的浮点除法：

```
# premtest.s - An example of using the FPREM1 instruction
.section .data
value1:
   .float 20.65
value2:
   .float 3.97
.section .bss
   .lcomm result, 4
.section .text
.globl _start
_start:
   nop
   finit
   flds value2
   flds value1
loop:
   fprem1
   fstsw %ax
   testb $4, %ah
   jnz loop

   fsts result
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

因为FPREM1指令是一个迭代的处理过程，所以不能保证它经过第一次执行就得到最终答案。TEST指令用于检查C2条件位的值（先使用FSTSW指令把它送到AX寄存器）。如果这一位被设置了，那么TEST指令就会生成非零值，并且JNZ指令会跳转回loop位置。当这一位被清空时，TEST指令生成零值，JNZ指令不会跳转，顺序执行。余数值存储在ST0寄存器中，使用FSTS指令把它复制到内存位置result。

汇编和连接程序之后，可以在调试器中运行它并且监视FPREM1指令如何确定余数值。对于前面显示的示例值，余数值应该如下：

```
(gdb) x/f &result
0x80490a8 <result>: 0.799999475
(gdb)
```

虽然余数值存储在ST0寄存器中，但是实际的商值没有存储在寄存器中。商值的最后3个有效位存储在控制寄存器中，使用控制寄存器中剩余的条件代码位，如下：

- 商位0在条件位1
- 商位1在条件位3
- 商位2在条件位0

必须手动地提取这些位以便构成商值的最低3位。

FPREM指令的输出显得有点儿奇怪，这是有原因的。在旧式80287 FPU协处理器的年代，FPTAN指令不能处理大于pi/4的角弧度。对于确定源角度值位于哪个象限中，FPREM指令是至关重要的。因为这涉及象限，所以只需要商的最低3位。从80387 FPU协处理器开始，FPTAN指令就没有这个限制了，并且FPREM指令的商值很难被用到。

### 9.3.3 三角函数

FPU的另一个巨大优势是计算三角函数的能力。一般的三角函数，比如正弦、余弦和正切，都可以轻松地从FPU获得。下面几节介绍在汇编语言程序中使用FPU三角函数。

1. FSIN和FCOS指令

在FPU中，基本的三角函数都按照相同的方式实现。这些指令都使用一个隐含的源操作数，它位于ST0寄存器中。当函数完成时，结果存放在ST0寄存器中。

这些函数唯一难于处理的地方在于它们都使用弧度作为源操作数的单位。如果正在处理的应用程序使用角度，那么在能够使用FPU的三角函数之前，必须把值转换为弧度。完成这一转换的公式如下：

```
radians = (degrees * pi) / 180
```

在FPU中，这一计算很容易通过下面的代码片断完成：

```
fsts degree1          # load the degrees value stored in memory into ST0
fidivs val180         # divide by the 180 value stored in memory
fldpi                 # load pi into ST0, degree/180 now in ST1
fmul %st(1), %st(0)   # multiply degree/180 and pi, saving in ST0
fsin                  # perform trig function on value in ST0
```

trigtest1.s程序演示这些函数：

```
# trigtest1.s - An example of using the FSIN and FCOS instructions
.section .data
degree1:
   .float 90.0
val180:
   .int 180
.section .bss
   .lcomm radian1, 4
   .lcomm result1, 4
   .lcomm result2, 4
.section .text
.globl _start
_start:
   nop
   finit
   flds degree1
   fidivs val180
   fldpi
   fmul %st(1), %st(0)
   fsts radian1
   fsin
   fsts result1
   flds radian1
   flds radian1
   fcos
   fsts result2

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

角度被转换为弧度之后，它被存储在内存位置radian1中。然后使用FSIN指令计算角的正弦

值，使用FCOS指令计算余弦值。程序运行之后，可以查看内存位置result1和result2中的结果：

```
(gdb) x/f &result1
0x80490bc <result1>:    1
(gdb) x/f &result2
0x80490c0 <result2>:    -4.37113883e-08
(gdb)
```

当然90度角的正弦是1，余弦是0。也可以使用其他角度值测试这个程序。

在产品型的程序中，预先计算pi/180的值并且把它存储在FPU中显然要快得多，而不是让处理器每次都计算这个值。

2. FSINCOS指令

如果你需要同时获得角的正弦值和余弦值，那么FSINCOS指令可以在一个简单的步骤内做到这一点。这条指令把正弦结果存放到ST0寄存器中，然后把余弦结果压入FPU寄存器中。这通常使余弦值存放在ST0中，正弦值存放在ST1中。trigtest2.s程序演示这条指令的使用：

```
# trigtest2.s - An example of using the FSINCOS instruction
.section .data
degree1:
   .float 90.0
val180:
   .int 180
.section .bss
   .lcomm sinresult, 4
   .lcomm cosresult, 4
.section .text
.globl _start
_start:
   nop
   finit
   flds degree1
   fidivs val180
   fldpi
   fmul %st(1), %st(0)
   fsincos
   fstps cosresult
   fsts sinresult

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

trigtest2.s程序的结果可以在内存位置cosresult和sinresult中看到：

```
(gdb) x/f &cosresult
0x80490b0 <cosresult>:  -2.71050543e-20
(gdb) x/f &sinresult
0x80490ac <sinresult>:  1
(gdb)
```

cosresult的值并不精确地等于0，但是已经非常接近了。sinresult是正确的值1。

3. FPTAN和FPATAN指令

FPTAN和FPATAN指令与它们对应的正弦和余弦指令有些不同。虽然它们计算正切和反正切

三角函数，但是它们的输入和输出需求稍有不同。

FPTAN指令使用标准的位于ST0寄存器中的隐含操作数（同样，角的单位必须是弧度，不能是角度）。正如我们预期的，正切值被计算出来并且存放到ST0寄存器。之后，值1.0被压入FPU堆栈，把正切值向下移动到ST1寄存器。

这样做的原因是为了向下兼容为80287 FPU协处理器编写的应用程序。在那个时期，FSIN和FCOS还不可用，计算这些值需要使用正切值的倒数。通过在FPTAN指令之后使用简单的FDIVR指令，就可以计算出余切值。

FPATAN指令使用两个隐含的源操作数。它计算角值ST1/ST0的反正切值，并且把结果存放在ST1中，然后弹出FPU堆栈，把值移动到ST0。这种形式用于支持计算无穷比例——就是说，当ST0为零时——的反正切。标准的ANSI C函数atan2(double x，double y)也使用相同的概念。

### 9.3.4 对数函数

FPU的对数函数提供用于执行底数为2的对数计算的指令。FYL2X指令执行如下计算：

```
ST(1) * log2 (ST(0))
```

FYL2X1指令执行以下这样的计算：

```
ST(1) * log2 (ST(0) + 1.0)
```

FSCALE指令计算ST(0)乘以2的ST(1)次乘方。它可以用于放大（通过在ST(1)中使用正值），也可以用于缩小（通过在ST(1)中使用负值）。fscaletest.s程序演示这个规则：

```
# fscaletest.s - An example of the FSCALE instruction
.section .data
value:
   .float 10.0
scale1:
   .float 2.0
scale2:
   .float -2.0
.section .bss
   .lcomm result1, 4
   .lcomm result2, 4
.section .text
.globl _start
_start:
   nop
   finit
   flds scale1
   flds value
   fscale
   fsts result1

   flds scale2
   flds value
   fscale
   fsts result2

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

第一个比例值（设置为2.0）被加载到ST(0)寄存器中，然后加载测试值（10.0）（使比例值移动到它应该在的位置ST(1)）。FSCALE指令执行之后，测试值乘以2的比例值次方，结果是测试值乘以4。

下面，第二个比例值（设置为-2.0）被加载，然后是测试值，再次执行FSCALE指令。这次负比例因子使测试值除以4。

汇编和连接程序之后，可以单步执行指令并且显示内存位置result1和result2中的结果：

```
(gdb) x/f &result1
0x80490b8 <result1>:    40
(gdb) x/f &result2
0x80490bc <result2>:    2.5
(gdb)
```

生成的值是我们期望的。

注意，FSCALE指令提供了乘以或者除以2的浮点值次方的便捷方式，这和第8章中对整数使用移位指令的效果类似。

虽然FPU的对数函数只提供以2为底数的对数运算，但是可以使用其他对数底数执行计算。为了使用底数为2的对数来计算其他底数的对数，可以使用下面的方程式：

```
log (base b) X = (1/log(base 2) b) * log(base 2) X
```

这可以使用FYL2X指令很容易地实现。logtest.s程序计算内存值的底数为10的对数：

```
# logtest.s - An example of using the FYL2X instruction
.section .data
value:
   .float 12.0
base:
   .float 10.0
.section .bss
   .lcomm result, 4
.section .text
.globl _start
_start:
   nop
   finit
   fld1
   flds base
   fyl2x
   fld1
   fdivp
   flds value
   fyl2x
   fsts result

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

logtest.s程序使用前面的方程式执行值12.0的底数为10的对数计算。程序一开始，把1.0加载到FPU寄存器中（第一个对数函数的Y值），然后加载底数值（10.0），执行这个值的底数为2的

对数计算。这样生成方程式前半部分的值（注意在这个例子中，因为选择底数为10，所以可以使用FLDL2T一条指令把值加载到ST（0）中）。这个值成为下一条FYL2X指令的新的Y值，X值为原始值（12.0）。最终结果应该等于12的底数为10的对数，即1.079 181 19。

*（译者注：上一段文字似乎不对，程序、说明与公式不符，可能是原书错了。程序基本上符合公式，但是在fsts指令之前似乎缺少一个乘法。如果在fsts指令之前加上乘法，上一段文字可以修改为：logtest.s程序使用前面的方程式计算值12.0的底数为10的对数。程序一开始，把1.0加载到FPU寄存器中（用于求第一个对数函数的倒数），然后加载底数值（10.0），计算这个值的底数为2的对数。然后求对数值的倒数，这样生成方程式前半部分的值（$1/\log_2 10$）。然后下一条FYL2X指令计算$\log_{10} 12$。最终结果应该等于12的底数为10的对数，即1.07918119。）*

汇编和连接程序之后，可以在调试器中运行程序并且查看内存位置result来检查结果：

```
(gdb) x/f &result
0x80490a8 <result>: 1.07918119
(gdb)
```

正确，logtest.s程序生成了12的底数为10的对数的正确结果。

## 9.4 浮点条件分支

不幸的是，浮点数的比较不像整数的比较那么容易。在处理整数时，很容易使用CMP指令并且评估EFLAGS寄存器中的值来确定值是大于、等于还是小于。

对于浮点数，不能奢望使用CMP指令。相反，FPU提供一些它自己的指令来比较浮点值。

### 9.4.1 FCOM指令系列

FCOM指令系列用于在FPU中比较两个浮点值。指令比较的一方是加载到FPU寄存器ST0中的值，另一方要么是另一个FPU寄存器，要么是内存中的浮点值。还有在比较之后把一个值或者两个值弹出FPU堆栈的选项。下表介绍可以使用的不同指令版本。

| 指 令 | 描 述 |
|---|---|
| FCOM | 比较ST0寄存器和ST1寄存器 |
| FCOM ST（x） | 比较ST0寄存器和另一个FPU寄存器 |
| FCOM source | 比较ST0寄存器和32位或者64位的内存值 |
| FCOMP | 比较ST0寄存器和ST1寄存器，并且弹出堆栈 |
| FCOMP ST（x） | 比较ST0寄存器和另一个FPU寄存器，并且弹出堆栈 |
| FCOMP source | 比较ST0寄存器和32位或者64位的内存值，并且弹出堆栈 |
| FCOMPP | 比较ST0寄存器和ST1寄存器，并且两次弹出堆栈 |
| FTST | 比较ST0寄存器和值0.0 |

比较的结果设置在状态寄存器的C0、C2和C3条件代码位中。比较可能产生的值列在下表中。

| 条 件 | C3 | C2 | C0 |
|---|---|---|---|
| ST0>source | 0 | 0 | 0 |
| ST0<source | 0 | 0 | 1 |
| ST0=source | 1 | 0 | 0 |

必须使用FSTSW指令把状态寄存器的值复制到AX寄存器或者内存位置中，然后使用TEST指令判断比较的结果。

fcomtest.s程序演示这一规则：

```
# fcomtest.s - An example of the FCOM instruction
.section .data
value1:
   .float 10.923
value2:
   .float 4.5532
.section .text
.globl _start
_start:
   nop
   flds value1
   fcoms value2
   fstsw
   sahf
   ja greater
   jb lessthan
   movl $1, %eax
   movl $0, %ebx
   int $0x80
greater:
   movl $1, %eax
   movl $2, %ebx
   int $0x80
lessthan:
   movl $1, %eax
   movl $1, %ebx
   int $0x80
```

在使用FSTSW指令获得FPU状态寄存器的值并且保存到AX寄存器之后，fcomtest.s程序使用一些技巧来判断FCOM指令的结果。SAHF指令用于把AH寄存器中的值加载到EFLAGS寄存器中。

SAHF指令把AH寄存器的第0、2、4、6和7位分别传送到进位、奇偶校验、对准、零和符号标志，不影响EFLAGS寄存器中的其他位。AH寄存器中的这些位正好包含FPU状态寄存器的条件代码值（感谢Intel的软件工程师）。FSTSW和SAHF指令的组合传送如下值：

- 把C0位传送到EFLAGS的进位标志
- 把C2位传送到EFLAGS的奇偶校验标志
- 把C3位传送到EFLAGS的零标志

传送完成之后，EFLAGS的进位、奇偶校验和零标志具有C0、C2和C3条件代码位的值，这样就为使用JA、JB和JZ指令确定两个浮点值的比较结果做了很好的转换工作。

根据内存中设置的值，fcomtest.s程序生成不同的结果代码。可以使用echo命令查看结果代码：

```
$ ./fcomtest
$ echo $?
2
$
```

结果代码为2表示第一个值（存储在内存位置value1中）大于第二个值（存储在内存位置

value2中)。可以改变程序中的值来确保比较功能完全正常。

关于相等性比较的一点说明：要记住，当把浮点值加载到FPU寄存器中时，它被转换为扩展双精度浮点值。这一处理可能导致一些舍入错误。单精度或者双精度值在加载到FPU寄存器之后有可能不等于原始值。检测浮点值的完全相等性不是个好主意，而要检测它们是否在预期值的小误差之内。

### 9.4.2 FCOMI指令系列

读者也许奇怪，既然在比较指令之后使用FSTSW和SAHF指令组合很有用，为什么不把它们合并成单一指令呢？回答是，确实这样做了。从奔腾Pro处理器系列开始，FCOMI指令可以完成这个任务。FCOMI指令系列执行浮点比较并且把结果存放到EFLAGS寄存器中的进位、奇偶校验和零标志。

下表介绍FCOMI指令系列。

| 指 令 | 描 述 |
|---|---|
| FCOMI | 比较ST0寄存器和ST（x）寄存器 |
| FCOMIP | 比较ST0寄存器和ST（x）寄存器，并且弹出堆栈 |
| FUCOMI | 在比较之前检查无序值 |
| FUCOMIP | 在比较之前检查无序值，并且在比较之后弹出堆栈 |

从上表的描述可以看出，FCOMI指令系列的局限性在于它们只能比较FPU寄存器中的两个值，不能比较FPU寄存器和内存中的值。

上表中的最后两条指令执行FCOM指令系列没有提供的功能。FUCOMI和FUCOMIP指令确保被比较的值是合法的浮点数（使用FPU标记寄存器）。如果出现无序值，就会抛出异常。

FCOMI指令的输出使用EFLAGS寄存器，如下表所示。

| 条件 | ZF | PF | CF |
|---|---|---|---|
| ST0>ST (x) | 0 | 0 | 0 |
| ST0<ST (x) | 0 | 0 | 1 |
| ST0=ST (x) | 1 | 0 | 0 |

为了验证FCOMI指令的功能，下面的fcomitest.s程序重现fcomtest.s程序的操作，但是这次使用FCOMI指令：

```
# fcomitest.s - An example of the FCOMI instruction
.section .data
value1:
   .float 10.923
value2:
   .float 4.5532
.section .text
.globl _start
_start:
   nop
   flds value2
   flds value1
```

```
    fcomi %st(1), %st(0)
    ja greater
    jb lessthan
    movl $1, %eax
    movl $0, %ebx
    int $0x80
greater:
    movl $1, %eax
    movl $2, %ebx
    int $0x80
lessthan:
    movl $1, %eax
    movl $1, %ebx
    int $0x80
```

因为FCOMI指令要求两个值都位于FPU寄存器中，所以按照相反的顺序加载它们，使得在进行比较时值value1在ST0中。汇编和连接程序之后，可以运行它并且查看结果代码:

```
$ ./fcomitest
$ echo $?
2
$
```

FCOMI指令生成和FCOM指令测试程序相同的结果。同样，可以改变测试值以便确保程序代码确实对不同比较情况都能生成正确的结果。

### 9.4.3 FCMOV指令系列

和用于整数的CMOV指令类似，FCMOV指令可以编写浮点值的条件传送。根据EFLAGS寄存器中的值，FCMOV系列的指令把FPU寄存器ST(x)中的源操作数传送到FPU寄存器ST(0)中的目标操作数。如果条件为true，就把ST(x)寄存器中的值传送到ST(0)寄存器。

因为根据EFLAGS寄存器进行传送，所以更常见的方式是在FCMOV指令之前使用FCOMI指令。

下表概述FCMOV指令系列。

| 指　令 | 描　述 |
|---|---|
| FCMOVB | 如果ST（0）小于ST（x），则进行传送 |
| FCMOVE | 如果ST（0）等于ST（x），则进行传送 |
| FCMOVBE | 如果ST（0）小于或者等于ST（x），则进行传送 |
| FCMOVU | 如果ST（0）无序，则进行传送 |
| FCMOVNB | 如果ST（0）不小于ST（x），则进行传送 |
| FCMOVNE | 如果ST（0）不等于ST（x），则进行传送 |
| FCMOVNBE | 如果ST（0）不小于或者等于ST（x），则进行传送 |
| FCMOVNU | 如果ST（0）非无序，则进行传送 |

指令的GNU格式是:

```
fcmovxx source, destination
```

其中source是ST(x)寄存器，destination是ST(0)寄存器。

fcmovtest.s程序演示传送的一些例子:

```
# fcmovtest.s - An example of the FCMOV xx instructions
.section .data
value1:
   .float 20.5
value2:
   .float 10.90
.section .text
.globl _start
_start:
   nop
   finit
   flds value1
   flds value2
   fcomi %st(1), %st(0)
   fcmovb %st(1), %st(0)

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

值被加载到FPU寄存器中（ST0＝10.90，ST1＝20.5）。FCOMI指令根据ST0和ST1中的值设置EFLAGS寄存器。如果ST0的值小于ST1的值（这个例子中就是这种情况），FCMOVB指令就把ST1中的值传送到ST0。

汇编和连接程序之后，可以在程序运行时检查FPU寄存器的值如何变化。FCMOVB指令执行之后，ST0和ST1寄存器都应该包含值20.5。

FCMOV指令在奔腾Pro和更新的处理器上是可用的。这些指令不能在更早期的IA-32处理器上工作。

## 9.5 保存和恢复FPU状态

不幸的是，在现在的IA-32处理器中，FPU数据寄存器必须完成双重工作。MMX技术使用FPU数据寄存器作为MMX数据寄存器，存储80位打包整数值用于计算。如果在同一个程序中使用FPU和MMX功能，就有可能“破坏”数据寄存器。

为了帮助防止这种情况，IA-32平台包含几个指令，它们可以保存FPU处理器的状态并且在其他处理完成之后恢复先前的状态。本节介绍可以用于保存和恢复FPU处理器状态的不同指令。

### 9.5.1 保存和恢复FPU环境

FSTENV指令用于把FPU的环境存储到一个内存块中。下面的FPU寄存器被存储:

- 控制寄存器
- 状态寄存器
- 标记寄存器
- FPU指令指针偏移量
- FPU数据指针
- FPU最后执行的操作码

这些值存储在一个28字节的内存块中。FLDENV指令用于把内存块的值加载回FPU环境中。fpuenv.s程序演示这些指令：

```
# fpuenv.s - An example of the FSTENV and FLDENV instructions
.section .data
value1:
   .float 12.34
value2:
   .float 56.789
rup:
   .byte 0x7f, 0x0b
.section .bss
   .lcomm buffer, 28
.section .text
.globl _start
_start:
   nop
   finit
   flds value1
   flds value2
   fldcw rup
   fstenv buffer

   finit
   flds value2
   flds value1

   fldenv buffer

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

fpuenv.s程序初始化FPU，把几个值加载到FPU的数据寄存器中，修改控制寄存器中的舍入位，然后把结果存储到内存位置buffer中。如果在FSTENV指令之后查看buffer位置，它应该如下：

```
(gdb) x/28b &buffer
0x80490c0 <buffer>:    0x7f   0x0b   0xff   0xff   0x00   0x30   0xff   0xff
0x80490c8 <buffer+8>:  0xff   0x0f   0xff   0xff   0x7e   0x80   0x04   0x08
0x80490d0 <buffer+16>: 0x23   0x00   0x00   0x00   0xb8   0x90   0x04   0x08
0x80490d8 <buffer+24>: 0x2b   0x00   0xff   0xff
(gdb)
```

读者也许会注意到内存位置中的控制寄存器（0x7f 0x0b）和状态寄存器（0x00 0x30）。存储FPU环境之后，FPU被初始化，然后其他几个值被存储到FPU数据寄存器中。使用info all命令查看FPU寄存器。

然后使用FLDENV指令从缓冲区恢复FPU环境。恢复之后，查看FPU中的寄存器。注意，FPU数据寄存器没有恢复到它们先前的值，但是控制寄存器的舍入位再次被设置为向上舍入。

### 9.5.2 保存和恢复FPU状态

FSTENV指令存储FPU环境，但是从前面的程序设计例子可以看出，FPU中的数据没有被保存。为了保存包括数据在内的完整FPU环境，必须使用FSAVE指令。

FSAVE指令把所有FPU寄存器复制到一个108字节的内存位置，然后初始化FPU状态。使用FRSTOR指令恢复FPU时，所有FPU寄存器（包括数据寄存器）都被恢复为执行FSAVE指令时的状态：

```
# fpusave.s - An example of the FSAVE and FRSTOR instructions
.section .data
value1:
   .float 12.34
value2:
   .float 56.789
rup:
   .byte 0x7f, 0x0b
.section .bss
   .lcomm buffer, 108
.section .text
.globl _start
_start:
   nop
   finit
   flds value1
   flds value2
   fldcw rup
   fsave buffer

   flds value2
   flds value1

   frstor buffer

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

把几个值加载到FPU数据寄存器并且设置舍入位之后，使用FSAVE指令把FPU状态存储在缓冲区位置。在FSAVE指令执行之前，可以使用调试器info all命令查看FPU的状态：

```
(gdb) info all
.
.
st0            56.78900146484375          (raw 0x4004e327f00000000000)
st1            12.340000152587890625      (raw 0x4002c570a40000000000)
st2            0          (raw 0x00000000000000000000)
st3            0          (raw 0x00000000000000000000)
st4            0          (raw 0x00000000000000000000)
st5            0          (raw 0x00000000000000000000)
st6            0          (raw 0x00000000000000000000)
st7            0          (raw 0x00000000000000000000)
fctrl          0xb7f      2943
fstat          0x3000     12288
ftag           0xfff      4095
fiseg          0x23       35
fioff          0x804807e          134512766
foseg          0x2b       43
fooff          0x80490b4          134516916
fop            0x0        0
(gdb)
```

在前面的清单中，可以看到两个数据值和新的控制寄存器设置。执行FSAVE指令之后，可以查看新的FPU状态：

```
(gdb) info all
.
.
st0            0          (raw 0x00000000000000000000)
st1            0          (raw 0x00000000000000000000)
st2            0          (raw 0x00000000000000000000)
st3            0          (raw 0x00000000000000000000)
st4            0          (raw 0x00000000000000000000)
st5            0          (raw 0x00000000000000000000)
st6            56.78900146484375          (raw 0x4004e327f00000000000)
st7            12.340000152587890625      (raw 0x4002c570a40000000000)
fctrl          0x37f      895
fstat          0x0        0
ftag           0xffff     65535
fiseg          0x0        0
fioff          0x0        0
foseg          0x0        0
fooff          0x0        0
fop            0x0        0
(gdb)
```

注意，堆栈顶部的值被移动了，使堆栈顶部原来的值现在位于寄存器堆栈的底部。并且，控制寄存器的值被复位为默认值。可以在调试器中查看缓冲区内存位置中的值的情况：

```
(gdb) x/108b &buffer
0x80490c0 <buffer>:    0x7f   0x0b   0xff   0xff   0x00   0x30   0xff   0xff
0x80490c8 <buffer+8>:  0xff   0x0f   0xff   0xff   0x7e   0x80   0x04   0x08
0x80490d0 <buffer+16>: 0x23   0x00   0x00   0x00   0xb4   0x90   0x04   0x08
0x80490d8 <buffer+24>: 0x2b   0x00   0xff   0xff   0x00   0x00   0x00   0x00
0x80490e0 <buffer+32>: 0x00   0xf0   0x27   0xe3   0x04   0x40   0x00   0x00
0x80490e8 <buffer+40>: 0x00   0x00   0x00   0xa4   0x70   0xc5   0x02   0x40
0x80490f0 <buffer+48>: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x80490f8 <buffer+56>: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x8049100 <buffer+64>: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x8049108 <buffer+72>: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x8049110 <buffer+80>: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x8049118 <buffer+88>: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x8049120 <buffer+96>: 0x00   0x00   0x00   0x00   0x00   0x00   0x00   0x00
0x8049128 <buffer+104> 0x00   0x00   0x00   0x00
(gdb)
```

缓冲区不仅包含控制、状态和标记寄存器，还包含FPU数据寄存器的值。执行FRSTOR指令之后，可以查看所有寄存器，它们都被恢复为执行FSAVE指令时的状态：

```
(gdb) info all
.
.
st0            56.78900146484375          (raw 0x4004e327f00000000000)
st1            12.340000152587890625      (raw 0x4002c570a40000000000)
st2            0          (raw 0x00000000000000000000)
st3            0          (raw 0x00000000000000000000)
st4            0          (raw 0x00000000000000000000)
```

```
st5            0          (raw 0x00000000000000000000)
st6            0          (raw 0x00000000000000000000)
st7            0          (raw 0x00000000000000000000)
fctrl          0xb7f      2943
fstat          0x3000     12288
ftag           0xfff      4095
fiseg          0x23       35
fioff          0x804807e          134512766
foseg          0x2b       43
fooff          0x80490b4          134516916
fop            0x0        0
(gdb)
```

## 9.6 等待和非等待指令

如果读者研究Intel的手册，可能会注意到一些浮点指令有对应的非等待版本。术语“等待”和“非等待”涉及指令如何处理浮点异常。

浮点异常在前面的“状态寄存器”小节中讨论过。浮点指令可能生成6种浮点异常。它们通常表明运算过程中出现了某些错误（比如试图以零作为除数）。

大多数浮点指令在执行之前必须等待以便确保前面的指令没有抛出异常。如果出现异常，在能够执行下一条指令之前必须先处理异常。

还有另一种方式，一些指令包含非等待版本，它们不等待浮点异常的检查。这些指令允许程序保存或者复位当前的FPU状态，而不处理任何悬而未决的异常。下表介绍可以使用的非等待指令。

| 指 令 | 描 述 |
|---|---|
| FNCLEX | 清空浮点异常标志 |
| FNSAVE | 把FPU状态保存到内存中 |
| FNSTCW | 保存FPU控制寄存器 |
| FNSTENV | 把FPU操作环境保存到内存中 |
| FNSTSW | 把FPU状态寄存器保存到内存或者AX寄存器中 |

## 9.7 优化浮点运算

浮点运算可能是汇编语言应用程序中最为耗费时间的部分。一定要尝试优化浮点代码，尽可能地提高运算的性能。

Intel提供了编写浮点程序的一些简单技巧：

- 确保浮点值不会上溢或者下溢出数据元素。
- 把精度控制位设置为单精度。
- 使用查找表实现简单的三角函数。
- 在可能的情况下，断开依赖链。例如，不计算z=a+b+c+d，而是计算x=a+b；y=c+d；z=x+y。

- 在FPU寄存器中尽可能多地保留方程式的值。
- 在处理整数和浮点值时，把整数加载到FPU寄存器中并且执行运算，这样比对整数使用浮点指令要快。例如，不使用FIDIV，而是使用FILD加载整数，然后对FPU寄存器中的值执行FDIVP指令。
- 尽可能使用FCOMI指令，不使用FCOM指令。

## 9.8 小结

本章讨论IA-32平台中FPU浮点运算功能。首先回顾以前讲过的FPU环境，它介绍FPU数据寄存器（它们组合在一起构成堆栈）、状态寄存器（它维护FPU的操作状态）、控制寄存器（它提供控制FPU的操作的方法）和标记寄存器（它是确定FPU数据寄存器状态的简单方式）。

简要地回顾FPU环境之后，本章介绍FPU的基本运算，包括执行简单浮点加法、减法、乘法和除法的指令。每条指令有6个版本，提供使用FPU数据寄存器和内存中的操作数的方法，还有使用整数和浮点值执行运算的指令。还演示了如何通过把所有值都保存在FPU寄存器中执行复杂数学方程式计算来提高性能。

下一节讲解高级浮点运算功能。首先讨论把浮点值从一种形式转换为另一种形式的功能（比如求绝对值和改变符号的指令）。然后，讨论用于计算部分余数的指令，包括如何计算浮点部分余数，以及如何使用FPU状态寄存器的条件代码位显示运算的结果。然后，学习了三角函数。FPU提供了所有基本的三角函数：FSIN、FCOS和FPTAN。在使用FPU三角函数时，要记住所有角值的单位都必须是弧度，这一点很重要。本章介绍了把角度转换为弧度的简单方法，以及如何在程序中使用这一方法。最后介绍了FPU对数函数，并且演示如何使用它们计算任意底数的对数。

接下来介绍FPU条件分支指令。与整数条件分支类似，FPU提供了可以在浮点应用程序中根据浮点变量的值创建分支的指令。FCOM指令使用状态寄存器的条件代码位表示两个变量是相等、小于还是大于。可以使用FSTSW和SAHF指令把条件代码位加载到标准EFLAGS寄存器中以便执行比较分支。较新的IA-32处理器还包含FCOMI指令，它执行比较并且自动地把条件代码位加载到EFLAGS寄存器中，使用EFLAGS寄存器的进位、奇偶校验和零标志作为指示符。最后介绍的FCMOV指令系列是非常好的工具，它根据比较的结果在FPU中传送值，不必执行分支指令。由于不破坏处理器的指令预取缓存，这能够显著地提高性能。

读者还学习了如何存储和获得FPU环境和状态。因为FPU和新的MMX技术共享FPU的资源，所以同时使用这两种技术的程序必须能够存储和恢复FPU的值。FPU环境由控制和状态寄存器以及FPU指令和数据指针构成，可以使用FSTENV指令把它们存储在一个28字节的内存位置中，然后可以在任何时候使用FLDENV指令获得它们。如果还需要存储FPU数据寄存器的值，FSAVE指令可以保存FPU环境和所有数据寄存器。保存所有这些值需要108字节的内存位置。但是，当使用FSAVE指令时要谨慎，因为指令完成之后会重新初始化FPU状态。会丢失控制寄存器中的所有设置。以后在任何时候，都可以使用FRSTOR指令把FPU状态恢复为执行FSAVE指令时的原始状态（包含数据值）。

本章还有两个简短的小节，介绍等待和非等待指令调用，以及如何优化浮点程序。对于每条FPU指令，都有可能出现错误。在执行下一条指令之前，FPU通常试图等待错误的出现。但是，在某些情况下可能不希望等待，比如试图在异常出现之前保存FPU状态的时候。几个FPU指令（都以FN开头）可以在不等待任何FPU异常的情况下执行。Intel文档还提供了在FPU环境中进行程序设计时要牢记的一些基本技巧。如果开发的应用程序需要尽可能高的处理速度，那么遵守这些浮点优化技巧是个好主意。

下一章将离开数学领域（终于完成了），进入字符串的领域。虽然处理器对字符串不特别感兴趣，但是人们是离不开它们的。为了使我们更轻松些，Intel提供了一些指令来帮助在处理器中操纵字符串值。这些内容都在下一章中介绍。

# 第10章 处理字符串

对于人类来说，文字的沟通方式是至关重要的。不幸的是，计算机没有被专门设计为用于和人类进行轻松的沟通。大多数高级语言提供专门的函数帮助程序员编写和人类进行交互的程序。作为汇编语言程序员，不能奢望使用这些函数。编写使用人类的语言进行交互的程序代码是程序员的责任。

字符串的使用帮助程序使用人类的语言和人类进行通信。虽然在汇编语言程序设计中使用字符串不是简单的事情，但并不是不可能的。本章指导读者如何处理字符串。第一节介绍通过把字符串从一个内存位置复制到另一个内存位置，从而在内存中传送字符串的指令。下一节介绍如何比较字符串的相等性，这和整数类似。最后一节介绍用于在字符串中搜索字符或者字符串的指令。在文本中搜索特定字符时，这一特性很方便。

注意，本章中介绍的字符串指令也可以应用于非字符串数据。传送、修改和比较数字数据块的操作也可以通过IA-32的字符串指令来完成。

## 10.1 传送字符串

处理字符串时最有用的功能之一就是，把字符串从一个内存位置复制到另一个内存位置的能力。如果回忆第5章“传送数据”中的内容，读者会记得不能使用MOV指令把数据从一个内存位置传送到另一个内存位置。

幸运的是，Intel创建了用于处理字符串数据的完整的指令系列。本节介绍IA-32的字符串传送指令，并且演示如何在程序中使用它们。

### 10.1.1 MOVS指令

创建MOVS指令是为了向程序员提供把字符串从一个内存位置传送到另一个内存位置的简单途径。MOVS指令有3种格式：

- MOVSB：传送单一字节
- MOVSW：传送一个字（2字节）
- MOVSL：传送一个双字（4字节）

Intel文档使用MOVSD传送双字。GNU汇编器决定使用MOVSL。

MOVS指令使用隐含的源和目标操作数。隐含的源操作数是ESI寄存器。它指向源字符串的内存位置。隐含的目标操作数是EDI寄存器。它指向字符串要被复制到的目标内存位置。记住操作数顺序的好方法是ESI中的S代表源（source），而EDI中的D代表目标（destination）。

使用GNU汇编器时，有两种方式加载ESI和EDI值。第一种方式是使用间接寻址（参见第5章）。通过在内存位置标签前面添加美元符号，内存位置的地址被加载到了ESI或者EDI寄存器中：

```
movl $output, %edi
```

这条指令把output标签的32位内存位置传送给EDI寄存器。

指定内存位置的另一种方式是LEA指令。LEA指令加载一个对象的有效地址。因为Linux使用32位值引用内存位置，所以对象的内存地址必须存储在32位的目标值中。源操作数必须指向一个内存位置，比如.data段中使用的标签。指令

```
leal output, %edi
```

把output标签的32位内存位置加载到EDI寄存器中。

下面的movstest1.s程序使用MOVS指令传送一些字符串：

```
# movstest1.s - An example of the MOVS instructions
.section .data
value1:
   .ascii "This is a test string.\n"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
_start:
nop
leal value1, %esi
leal output, %edi
movsb
movsw
movsl

movl $1, %eax
movl $0, %ebx
int $0x80
```

movstest1.s程序把内存位置value1的位置加载到ESI寄存器中，把output内存位置的位置加载到EDI寄存器中。当执行MOVSB指令时，它把1字节的数据从value1位置传送到output位置。因为在.bss段中声明了output变量，所以存放在这里的任何字符串数据的结尾会被自动地加上空字符。

有趣的事情是MOVSW指令执行时发生的情况。如果在调试器中运行程序，可以看到每条MOVS指令执行后的内存位置output：

```
(gdb) s
13          movsb
(gdb) s
14          movsw
(gdb) x/s &output
0x80490b0 <output>:      "T"
(gdb) s
15          movsl
(gdb) x/s &output
0x80490b0 <output>:      "Thi"
(gdb) s
17          movl $1, %eax
(gdb) x/s &output
0x80490b0 <output>:      "This is"
(gdb)
```

正如我们期望的，MOVSB指令把“T”从value1位置传送到output位置。但是，无须改变ESI和EDI寄存器，当运行MOVSW指令时，它没有传送“Th”（字符串的前2个字节），而是把“hi”从value1位置传送到了output位置。然后MOVSL指令继续添加下4个字节的值。产生这样的结果是有原因的。

每次执行MOVS指令时，数据传送后，ESI和EDI寄存器会自动改变，为另一次传送做准备。通常这是件好事儿，但是有时候会变得有些难于处理。

这一操作难于处理的部分之一就是寄存器向哪个方向改变。ESI和EDI寄存器可能自动地递增，也可能自动地递减，这取决于EFLAGS寄存器中的DF标志。

如果DF标志被清零，那么每条MOVS指令执行之后ESI和EDI寄存器就会递增。如果DF标志被设置，那么每条MOVS指令执行之后ESI和EDI寄存器就会递减。因为movstest1.s程序没有专门设置DF标志，我们遵守的是当前的设置。为了确保DF标志被设置为正确的方向，可以使用下面的命令：

- CLD用于将DF标志清零
- STD用于设置DF标志

使用STD指令时，ESI和EDI寄存器在每条MOVS指令执行之后递减，所以它们应该指向字符串的末尾，而不是开头。movstest2.s程序演示这一情况：

```
# movstest2.s - A second example of the MOVS instructions
.section .data
value1:
   .ascii "This is a test string.\n"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
_start:
   nop
   leal value1+22, %esi

   leal output+22, %edi

   std
   movsb
   movsw
   movsl

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

这一次，value1内存位置的地址位置被存放到EAX寄存器中，测试字符串的长度（减去1是因为字符串从地址0开始）与它相加，这个值被存放到ESI寄存器中。这使ESI寄存器指向测试字符串的末尾。对EDI进行相同的操作，使它指向内存位置output的末尾。STD指令用于设置DF标志，使ESI和EDI寄存器在每条MOVS指令执行之后递减。

3条MOVS指令在两个字符串位置之间传送1、2和4个字节的数据。但是这一次，结果是不同的。3条MOVS指令执行之后，可以使用调试器查看内存位置output：

```
(gdb) x/23b &output
0x80490b8 <output>:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x80490c0 <output+8>:  0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x80490c8 <output+16>: 0x00    0x00    0x00    0x6e    0x67    0x2e    0x0a
(gdb)
```

注意，output位置的字符串从字符串的末尾开始填充，但是3条MOVS指令执行之后，只有4个内存位置被填充了。在向前移动的movstest1.s程序中，使用相同的3条指令却填充了7个内存位置。为什么会这样?

答案和值如何被复制有关。尽管ESI和EDI寄存器向后计数，MOVSW和MOVSL指令还是按照向前的顺序获得内存位置。当MOVSB指令完成时，它使ESI和EDI寄存器递减1，但是MOVSW指令获得两个内存位置。同样，当MOVSW指令完成时，它使ESI和EDI寄存器递减2，但是MOVSL指令获得4个内存位置。图10-1演示这种情况。

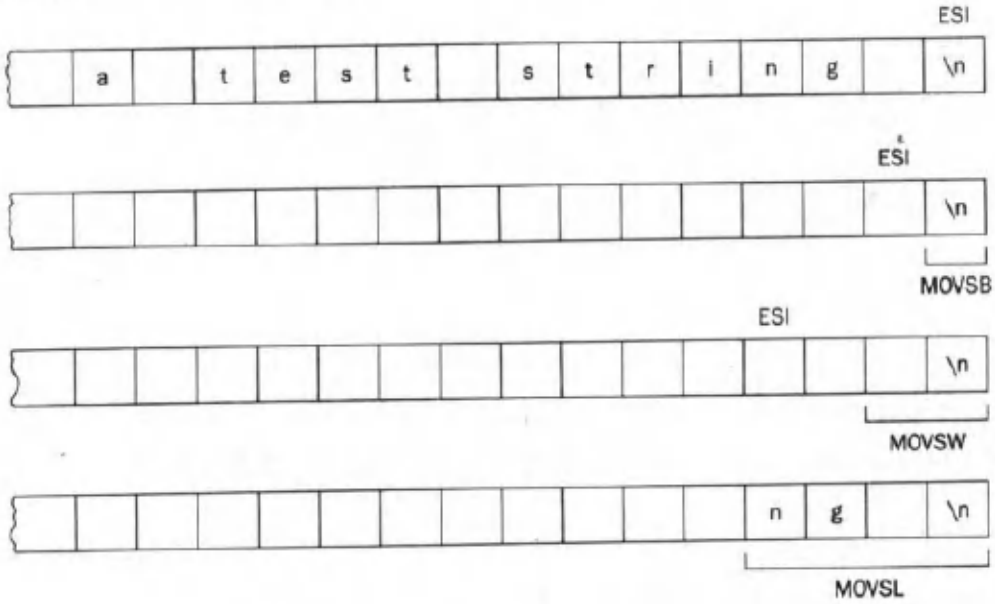



图 10-1

当然，解决这个问题的方法是使每条指令使用相同长度的块进行传送。如果所有3条指令都是MOVSW指令，那么ESI的值就会每次递减2，并且将传送2个字节。

要记住，如果使用STD指令向后处理字符串，MOVSW和MOVSL指令仍旧向前获取内存位置，这一点很重要。

如果要复制大型字符串，容易发现将使用很多MOVSL指令才能获得全部数据。为了使之更简单，可以试图把MOVSL指令放到循环中，通过把ECX寄存器设置为字符串的长度来进行控制。movstest3.s程序演示这种情况:

```
# movstest3.s - An example of moving an entire string
.section .data
value1:
   .ascii "This is a test string.\n"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
```

```
_start:
   nop
   leal value1, %esi
   leal output, %edi
   movl $23, %ecx
   cld
loop1:
   movsb
   loop loop1

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

像以前一样，ESI和EDI寄存器被设置为源和目标内存位置。ECX寄存器被设置为要传送的字符串的长度。循环部分持续地执行MOVSB指令，直到整个字符串传送完毕。查看内存位置output的字符串值可以看到这个结果：

```
(gdb) x/s &output
0x80490b0 <output>:      "This is a test string.\n"
(gdb)
```

虽然这种方法能够起作用，但是Intel提供了更加简单的方法：使用REP指令。

### 10.1.2 REP前缀

REP指令的特殊之处在于它自己不执行什么操作。这条指令用于按照特定次数重复执行字符串指令，由ECX寄存器中的值进行控制。这和使用循环类似，但是不需要额外的LOOP指令。REP指令重复地执行紧跟在它后面的字符串指令，直到ECX寄存器中的值为零。这就是为什么称它为前缀的原因。

1. 逐字节地传送字符串

MOVSB指令可以和REP指令一起使用，每次1字节地把字符串传送到另一个位置。reptest1.s程序演示如何这样操作：

```
# reptest1.s - An example of the REP instruction
.section .data
value1:
   .ascii "This is a test string.\n"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
_start:
   nop
   leal value1, %esi
   leal output, %edi
   movl $23, %ecx
   cld
   rep movsb

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

要传送的字符串长度被加载到ECX寄存器中，然后使用REP指令执行MOVSB指令23次（字符串的长度），每次传送1字节的数据。在调试器中单步执行程序时，REP指令仍然只被算作一个指令步骤，而不是23个。可以从调试器看到下面的输出：

```
$ gdb -q reptest1
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file reptest1.s, line 11.
(gdb) run
Starting program: /home/rich/palp/chap10/reptest1

Breakpoint 1, _start () at reptest1.s:11
11          leal value1, %esi
Current language:  auto; currently asm
(gdb) s
12          leal output, %edi
(gdb) s
13          movl $23, %ecx
(gdb) s
14          cld
(gdb) s
15          rep movsb
(gdb) s
17          movl $1, %eax
(gdb) x/s &output
0x80490b0 <output>:      "This is a test string.\n"
(gdb)
```

虽然单步执行指令时，REP指令只占用一个步骤，但是在这个步骤之后，源字符串的所有23个字节都被传送到了目标字符串位置。

2. 逐块地传送字符串

并不限于只能逐字节地传送字符串。也可以使用MOVSW和MOVSL指令在每次迭代中传送1字节以上的数据。

如果使用MOVSW和MOVSL指令，ECX寄存器就应该包含遍历字符串所需的迭代次数。例如，如果要传送8字节的字符串，如果使用MOVSB指令的话，就需要把ECX设置为8，使用MOVSW指令就设置为4，使用MOVSL指令就设置为2。

使用MOVSW和MOVSL指令遍历字符串时，小心不要超出字符串的边界。如果超出边界，请看一下reptest2.s程序中发生的情况：

```
# reptest2.s - An incorrect example of using the REP instruction
.section .data
value1:
   .ascii "This is a test string.\n"
value2:
   .ascii "Oops"
.section .bss
   .lcomm output, 23
.section .text
.globl _start
_start:
   nop
   leal value1, %esi
```

```
leal output, %edi
movl $6, %ecx
cld
rep movsl

movl $1, %eax
movl $0, %ebx
int $0x80
```

上面的例子自作聪明地试图只通过6次循环，传送每个4字节的数据块。唯一的问题在于源字符串的整个数据长度并不正好是4的倍数。最后一次执行MOVSL指令时，它不仅获得value1字符串的末尾，而且会错误地获得定义的下一个字符串的一字节的数据。可以从调试器的输出发现这一点：

```
$ gdb -q reptest2
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file reptest2.s, line 13.
(gdb) run
Starting program: /home/rich/palp/chap10/reptest2

Breakpoint 1, _start () at reptest2.s:13
13          leal value1, %esi
Current language:  auto; currently asm
(gdb) s
14          leal output, %edi
(gdb) s
15          movl $6, %ecx
(gdb) s
16          cld
(gdb) s
17          rep movsl
(gdb) s
19          movl $1, %eax
(gdb) x/s &output
0x80490b0 <output>:      "This is a test string.\nO"
(gdb)
```

现在output字符串的输出包含value2字符串的第一个字符，它被添加到了value1字符串中。这完全不是我们希望的结果。

3. 传送大型字符串

显然，尽可能多地使用MOVSL指令传送字符串的字符将使效率更高。但是如reptest2.s程序中演示的，问题在于你必须知道什么时候停止使用MOVSL指令并且转回使用MOVSB指令。技巧在于匹配字符串的长度。

当知道字符串的长度时，就容易执行整数除法（参见第8章）以便确定字符串中包含多少个双字。然后余数可以使用MOVSB指令进行传送（迭代次数应该小于3次）。reptest3.s程序演示这种处理方式：

```
# reptest3.s - Moving a large string using MOVSL and MOVSB
.section .data
string1:
   .asciz "This is a test of the conversion program!\n"
```

```
length:
   .int 43
divisor:
   .int 4
.section .bss
   .lcomm buffer, 43
.section .text
.globl _start
_start:
   nop

   leal string1, %esi
   leal buffer, %edi
   movl length, %ecx
   shrl $2, %ecx

   cld
   rep movsl
   movl length, %ecx
   andl $3, %ecx
   rep movsb

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

和以往一样，reptest3.s程序把源和目标内存位置加载到ESI和EDI寄存器中，但是然后把字符串长度值加载到AX寄存器中。为了使字符串长度被4整除，使用SHR指令把长度值向右移动2位（这和被4整除相同），这把商值加载到ECX寄存器中。然后使REP MOVSL指令组合执行这个值指定的次数。完成之后，使用一般的数学技巧确定余数值。

如果除数是2的乘方（4就是），可以通过从除数中减去1，然后把它和被除数进行AND操作快速地获得余数。然后把这个值加载到ECX寄存器中，执行REP MOVSB指令组合来传送剩余的字符。

可以在调试器中监视程序的执行情况。首先，在执行REP MOVSL指令组合之后停止程序，显示内存位置buffer的内容：

```
(gdb) s
22          movl %edx, %ecx
(gdb) x/s &buffer
0x80490d8 <buffer>:     "This is a test of the conversion program"
(gdb)
```

注意，前40个字符从源字符串传送到了目标字符串。下面，执行REP MOVSB指令组合，并且再次查看内存位置buffer的内容：

```
(gdb) s
23          rep movsb
(gdb) s
25          movl $1, %eax
(gdb) x/s &buffer
0x80490d8 <buffer>:     "This is a test of the conversion program!\n"
(gdb)
```

字符串中的最后两个字符被成功地传送了。同样，应该尝试不同的字符串和字符串长度组合以便验证这种方法能够正确地工作。

4. 按照相反的顺序传送字符串

向后执行和向前执行REP指令都是可以的。可以把DF标志设置为对字符串进行向后处理，按照相反的方向在内存位置之间传送它。reptest4.s程序演示这种情况：

```
# reptest4.s - An example of using REP backwards
.section .data
value1:
   .asciz "This is a test string.\n"
.section .bss
   .lcomm output, 24
.section .text
.globl _start
_start:
   nop
   leal value1+22, %esi
   leal output+22, %edi
   movl $23, %ecx
   std
   rep movsb

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

和movstest2.s程序类似，reptest4.s程序把源和目标字符串的末尾的位置加载到ESI和EDI寄存器中，然后使用STD指令设置DF标志。这使目标字符串按照相反的顺序被存储（尽管无法通过查看调试器验证这一点，因为REP指令仍然在一个步骤中传送所有的字节）：

```
(gdb) s
20          std
(gdb) s
21          rep movsb
(gdb) s
23          movl $1, %eax
(gdb) x/s &output
0x80490c0 <output>:      "This is a test string.\n"
(gdb)
```

### 10.1.3 其他REP指令

虽然REP指令很方便，在处理字符串时，也可以使用它的几个其他版本。除了监视ECX寄存器的值之外，还有监视零标志（ZF）的状态的REP指令。下表介绍可以使用的其他REP指令。

| 指　令 | 描　述 |
|---|---|
| REPE | 等于时重复 |
| REPNE | 不等于时重复 |
| REPNZ | 不为零时重复 |
| REPZ | 为零时重复 |

REPE和REPZ指令是相同指令的同义词，REPNE和REPNZ指令是同义词。

虽然MOVS指令一般不使用这些REP指令，但是本章后面讨论的比较和扫描字符串函数广泛地利用了它们。

## 10.2 存储和加载字符串

除了把字符串从一个内存位置传送到另一个内存位置之外，还有用于把内存中的字符串值加载到寄存器中以及传送回内存位置中的指令。本节介绍用于这个目的的STOS和LODS指令。

### 10.2.1 LODS指令

LODS指令用于把内存中的字符串值传送到EAX寄存器中。和MOVS指令一样，LODS指令有3种不同格式：

- LODSB：把一个字节加载到AL寄存器中
- LODSW：把一个字（2字节）加载到AX寄存器中
- LODSL：把一个双字（4字节）加载到EAX寄存器中

Intel文档使用LODSD加载双字。GNU汇编器使用LODSL。

LODS指令使用ESI寄存器作为隐含的源操作数。ESI寄存器必须包含要加载的字符串所在的内存地址。数据传送完成之后，LODS指令按照加载的数据的数量递增或者递减（取决于DF标志状态）ESI寄存器。

本章后面介绍的STOS和SCAS指令都利用存储在EAX寄存器中的数据。LODS指令把字符串值存放到EAX寄存器中便于这些指令的操作。虽然可以使用REP指令重复执行LODS指令，但是很可能永远都不会这么做，因为能够加载到EAX寄存器中的数据最多是4个字节，这可以通过单一LODSL指令完成。

### 10.2.2 STOS指令

使用LODS指令把字符串值存放到EAX寄存器之后，可以使用STOS指令把它存放到另一个内存位置中。和LODS指令类似，根据要传送的数据的数量，STOS指令有3种格式：

- STOSB：存储AL寄存器中一个字节的数据
- STOSW：存储AX寄存器中一个字（2字节）的数据
- STOSL：存储EAX寄存器中一个双字（4个字节）的数据

STOS指令使用EDI寄存器作为隐含的目标操作数。执行STOS指令时，它按照使用的数据长度递增或者递减EDI寄存器的值。

STOS指令本身没有什么太令人兴奋的地方。仅仅把单一字节、字或者双字的字符串值存放到内存位置中不是很难的工作。STOS指令真正能够提供的方便是和REP指令一起使用，多次把一个字符串值复制到大型字符串值中的时候——例如，把空格字符（ASCII值0x20）复制到256字节的缓冲区区域。

stostest1.s程序演示这一概念：

```
# stostest1.s - An example of using the STOS instruction
```

```
.section .data
space:
   .ascii " "
.section .bss
   .lcomm buffer, 256
.section .text
.globl _start
_start:
   nop
   leal space, %esi
   leal buffer, %edi
   movl $256, %ecx
   cld
   lodsb
   rep stosb

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

stostest1.s程序把ASCII空格字符加载到AL寄存器中，然后把它复制到buffer标签指向的内存位置中256次。可以使用调试器查看指令执行之前和之后的内存位置buffer的值：

```
(gdb) s
15          lodsb
(gdb) s
16          rep stosb
(gdb) print/x $eax
$1 = 0x20
(gdb) x/10b &buffer
0x80490a0 <buffer>:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0x80490a8 <buffer+8>:   0x00    0x00
(gdb) s
18          movl $1, %eax
(gdb) x/10b &buffer
0x80490a0 <buffer>:     0x20    0x20    0x20    0x20    0x20    0x20    0x20    0x20
0x80490a8 <buffer+8>:   0x20    0x20
(gdb)
```

输出显示通过LODSB指令把空格字符加载到了AL寄存器中。在STOSB指令执行之前，内存位置buffer包含零（这是我们期望的，因为它是在.bss段中构建的）。STOSB指令执行之后，内存位置buffer包含的都是空格。

### 10.2.3 构建自己的字符串函数

STOS和LODS指令可以用于各种字符串操作。通过使ESI和EDI寄存器指向相同的字符串，可以对字符串执行简单的操作。可以使用LODS指令遍历字符串，一次把一个字符加载到AL寄存器中，对这个字符执行某些操作，然后使用STOS指令把新的字符加载回字符串中。

convert.s程序演示这个原理，它把ASCII字符串都转换为大写字母：

```
# convert.s - Converting lower to upper case
.section .data
string1:
   .asciz "This is a TEST, of the conversion program!\n"
```

```
length:
   .int 43
.section .text
.globl _start
_start:
   nop
   leal string1, %esi
   movl %esi, %edi
   movl length, %ecx
   cld
loop1:
   lodsb
   cmpb $'a', %al
   jl skip
   cmpb $'z', %al
   jg skip
   subb $0x20, %al
skip:
   stosb
   loop loop1
end:
   pushl $string1
   call printf
   addl $4, %esp
   pushl $0
   call exit
```

convert.s程序把内存位置string1加载到ESI和EDI寄存器中，把字符串长度加载到ECX寄存器中。然后程序使用LOOP指令对字符串中的每个字符执行字符检查。程序进行字符检查的方法是，把每个字符加载到AL寄存器中，并且判断它是否小于字母a的ASCII值（0x61），或者大于字母z的ASCII值（0x7a）。如果字符在这个范围之内，那么它必然是小写字母，必须通过减去0x20把它转换为大写字母。

不管是否对字符进行了转换，都必须把它存放回字符串，以便保持ESI和EDI寄存器的同步。对每个字符都运行STOSB指令，然后代码向后循环到下一个字符，直到完成字符串中所有字符的处理。

对程序进行汇编并且把它和C库连接到一起之后，可以运行它以便查看它是否正确工作：

```
$ ./convert
THIS IS A TEST, OF THE CONVERSION PROGRAM!
$
```

确实，它像我们期望的那样运行。读者可以在字符串中使用不同类型的ASCII字符来测试程序（记住改变长度值，使之匹配新的字符串）。

## 10.3 比较字符串

把字符串从一个位置传送到另一个位置是有用的，但是其他字符串函数可以真正地帮助解决处理字符串时出现的问题。可用的最有帮助的字符串功能之一就是比较字符串的能力。许多程序都需要比较用户的输入值，或者将字符串记录和搜索值比较。本节介绍在汇编语言程序中用于比较字符串值的方法。

### 10.3.1 CMPS指令

CMPS指令系列用于比较字符串值。和其他字符串指令一样，CMPS指令有3种格式：

- CMPSB：比较字节值
- CMPSW：比较字（2字节）值
- CMPSL：比较双字（4字节）值

和其他字符串指令一样，隐含的源和目标操作数的位置同样存储在ESI和EDI寄存器中。每次执行CMPS指令时，根据DF标志的设置，ESI和EDI寄存器按照被比较的数据的长度递增或者递减。

CMPS指令从源字符串中减去目标字符串，并且适当地设置EFLAGS寄存器的进位、符号、溢出、零、奇偶校验和辅助进位标志。CMPS指令执行之后，可以根据字符串的值，使用一般的条件跳转指令跳转到分支。

cmpstest1.s程序演示使用CMPS指令的简单例子：

```
# cmpstest1.s - A simple example of the CMPS instruction
.section .data
value1:
   .ascii "Test"
value2:
   .ascii "Test"
.section .text
.globl _start
_start:
   nop
   movl $1, %eax
   leal value1, %esi
   leal value2, %edi
   cld
   cmpsl
   je equal
   movl $1, %ebx
   int $0x80
equal:
   movl $0, %ebx
   int $0x80
```

cmptest1.s程序比较两个字符串值，并且根据比较的结果设置程序的返回代码。程序首先把exit系统调用值加载到EAX寄存器中。把要测试的两个字符串的位置加载到ESI和EDI寄存器中之后，cmpstest1.s程序使用CMPSL指令比较字符串的前4个字节。如果字符串相等，就使用JE指令跳转到标签equal，这里把程序结果代码设置为0并且退出。如果字符串不相等，则不会跳转到分支，程序顺序执行，设置结果代码为1并且退出。

汇编和连接程序之后，可以通过运行它并且查看结果代码来检测程序：

```
$ ./cmpstest1
$ echo $?
0
$
```

结果代码为0，表示字符串互相匹配。为了进行测试，可以改变一个字符串并且再次对程序进行汇编以便查看结果代码是否改变为1。

对于检测最长4个字符的字符串的匹配情况来说，这种技术很不错，但是对于更长的字符串该怎么办？答案是使用REP指令，这将在下一节中介绍。

### 10.3.2 CMPS和REP一起使用

REP指令可以用于跨越多个字节重复地进行字符串比较，但是这里有个问题。不幸的是，REP指令不在两个重复的过程之间检查标志的状态；记住，它只关心ECX寄存器中的计数值。

解决方案是使用REP指令系列中的其他指令：REPE、REPNE、REPZ和REPNZ。这些指令在每次重复过程中检查零标志，如果零标志被设置，就停止重复。这使得可以逐字节地检查字符串以便确定它们是否匹配。只要遇到不匹配的字符对，REP指令就会停止重复。

cmpstest2.s程序演示如何进行这样的比较：

```
# cmpstest2.s - An example of using the REPE CMPS instruction
.section .data
value1:
   .ascii "This is a test of the CMPS instructions"
value2:
   .ascii "This is a test of the CMPS Instructions"
.section .text
.globl _start
_start:
   nop
   movl $1, %eax
   lea value1, %esi
   leal value2, %edi
   movl $39, %ecx
   cld
   repe cmpsb
   je equal
   movl %ecx, %ebx
   int $0x80
equal:
   movl $0, %ebx
   int $0x80
```

cmpstest2.s程序把源和目标字符串的位置加载到ESI和EDI寄存器中，把字符串长度加载到ECX寄存器中。REPE CMPSB指令逐字节地重复字符串的比较，直到ECX寄存器为零，或者零标志被设置——这表明不匹配。

REPE指令执行之后，像以往那样使用JE指令检查EFLAGS寄存器以便确定字符串是否相等。如果REPE指令退出，则零标志将被设置，JE指令不跳转到分支，表示字符串不相同。ESI和EDI寄存器将包含字符串中不匹配字符的内存位置，并且ECX寄存器将包含不匹配字符在字符串中的位置（从字符串的末尾向回计数）。

这个例子还说明字符串比较是区分大小写的。两个字符串之间只有一个字符的大小写有区别，这会被比较程序检测到：

```
$ ./cmpstest2
$ echo $?
11
$
```

CMPS指令从源字符串的十六进制值中减去目标字符串的值。每个字符的ASCII代码是不同的，所以字符串之间的任何区别都会被检测出来。

### 10.3.3 字符串不等

在讨论比较字符串的主题时，探讨一下字符串不等的概念是个好主意。比较字符串时，容易确定两个字符串相等的情况。字符串“test”和“test”永远都是相等的。但是，和字符串“test1”进行比较呢？它应该小于还是大于字符串“test”？

在整数比较中，容易理解值100大于10，但是确定字符串不等不是这么简单的概念。试图确定字符串“less”是小于还是大于字符串“greater”并不容易理解。如果应用程序必须确定字符串不等，就必须使用专门的方法来确定。

最常用于比较字符串的方法称为词典式顺序（lexicographical order）。这通常被称为字典顺序（dictionary order），因为这是字典对词进行排序的标准。当翻开字典时，会发现词是如何排序的。词典式顺序的基本规则如下：

- 按字母表顺序，较低的字母小于较高的字母
- 大写字母小于小写字母

读者也许会注意到这些规则遵循ASCII字符编码值的标准。很容易把这些规则应用于长度相同的字符串。字符串“test”将小于字符串“west”，但是大于字符串“Test”。当处理长度不同的字符串时，要困难一些。

比较长度不同的两个字符串时，按照长度短一些的字符串中的字符数量进行比较。如果短字符串大于长字符串中相同数量的字符，那么短字符串就大于长字符串。如果短字符串小于长字符串中相同数量的字符，那么短字符串就小于长字符串。如果短字符串等于长字符串中相同数量的字符，那么长字符串就大于短字符串。

使用这一规则，下面的例子就为真：

- “test”大于“boomerang”
- “test”小于“velocity”
- “test”小于“test1”

strcomp.s程序演示字符串的不等和相等的比较：

```
# strcomp.s - An example of comparing strings
.section .data
string1:
   .ascii "test"
length1:
   .int 4
string2:
   .ascii "test1"
length2:
   .int 5
.section .text
.globl _start
_start:
   nop
   lea string1, %esi
```

```
    lea string2, %edi
    movl length1, %ecx
    movl length2, %eax
    cmpl %eax, %ecx
    ja longer
    xchg %ecx, %eax
longer:
    cld
    repe cmpsb
    je equal
    jg greater
less:
    movl $1, %eax
    movl $255, %ebx
    int $0x80
greater:
    movl $1, %eax
    movl $1, %ebx
    int $0x80
equal:
    movl length1, %ecx
    movl length2, %eax
    cmpl %ecx, %eax
    jg greater
    jl less
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

strcomp.s程序定义两个字符串string1和string2，还有它们的长度（length1和length2）。程序生成的结果代码反映两个字符串的比较情况，见下表。

| 结果代码 | 描　述 |
|---|---|
| 255 | string1小于string2 |
| 0 | string1等于string2 |
| 1 | string1大于string2 |

为了执行词典式顺序的比较，首先必须确定短字符串的长度。这是通过把两个字符串的长度加载到寄存器中，然后使用CMP指令比较它们完成的。短的长度值被加载到ECX寄存器中，供REPE指令使用。

下面，使用REPE和CMPSB指令，按照短字符串的长度逐字节地比较两个字符串。如果第一个字符串大于第二个字符串，程序就跳转到标签greater位置的分支，设置结果代码为1并且退出。如果第一个字符串小于第二个字符串，程序就顺序执行，设置结果代码为255并且退出。

如果两个字符串相等，就要做更多的工作——程序还必须确定两个字符串中哪个更长。如果第一个字符串更长，则它更大，程序就跳转到标签greater位置的分支，设置结果代码为1并且退出。如果第二个字符串更长，那么第一个字符串就较小，程序跳转到标签less位置的分支，设置结果代码为255并且退出。如果两个长度不大于也不小于，那么不仅两个字符串匹配，而且它们的长度也相等，所以字符串相等。

读者可以在string1和string2位置设置不同的字符串，以便测试词典式顺序的比较（记住相应

地改变字符串的长度值)。使用strcomp.s程序清单中列出的值，输出应该如下:

```
$ ./strcomp
$ echo $?
255
$
```

即第一个字符串“test”小于第二个字符串“test1”。

## 10.4 扫描字符串

有时候扫描字符串搜索特定字符或者字符序列是有用的。一种方法是使用LODS指令遍历字符串，把每个字符加载到AL寄存器中，然后把AL寄存器中的字符和搜索字符进行比较。这种方法当然能够工作，但将是很耗费时间的过程。

Intel提供了使用另一种字符串指令完成这一工作的更好的方式。SCAS指令提供了扫描字符串搜索特定的一个字符或者一组字符的方式。本节介绍如何在程序中使用SCAS指令。

### 10.4.1 SCAS指令

SCAS指令系列用于扫描字符串搜索一个或者多个字符。和其他字符串指令一样，SCAS指令有3个版本:

- SCASB: 比较内存中的一个字节和AL寄存器的值
- SCASW: 比较内存中的一个字和AX寄存器的值
- SCASL: 比较内存中的一个双字和EAX寄存器的值

SCAS指令使用EDI寄存器作为隐含的目标操作数。EDI寄存器必须包含要扫描的字符串的内存地址。和其他字符串指令一样，当执行SCAS指令时，EDI寄存器的值按照搜索字符的数据长度递增或者递减（这取决于DF标志的值)。

进行比较时，会相应地设置EFLAGS的辅助进位、进位、奇偶校验、溢出、符号和零标志。可以使用标准的条件分支指令检查扫描的结果。

SCAS指令本身没有什么令人兴奋的地方。它仅仅是把EDI寄存器当前指向的字符和AL寄存器中的字符进行比较，这和CMPS指令类似。把SCAS与REPE和REPNE前缀一起使用时，它的方便性才显现出来。

这两个前缀可以扫描整个字符串，查找特定的搜索字符（或者字符序列)。REPE和REPNE指令常常用于在找到搜索字符时停止扫描。但是，使用这两个指令时要谨慎，因为它们的行为也许和你想像的相反:

- REPE: 扫描字符串的字符，查找不匹配搜索字符的字符
- REPNE: 扫描字符串的字符，查找匹配搜索字符的字符

对于大多数字符串扫描，使用REPNE指令，因为它将在字符串中找到搜索字符时停止扫描。当找到字符时，EDI寄存器包含紧跟在定位到的字符后面的内存地址。这是因为REPNE指令在执行SCAS指令之后递增EDI寄存器。ECX寄存器包含搜索字符距离字符串末尾的位置。使用这个值时要谨慎，因为它是从字符串的末尾开始计数的。为了得到距离字符串开头的位置，要从

这个值减去字符串长度并且反转符号。

scastest1.s程序演示使用REPNE和SCAS指令在字符串中搜索一个字符:

```
# scastest1.s - An example of the SCAS instruction
.section .data
string1:
   .ascii "This is a test - a long text string to scan."
length:
   .int 44
string2:
   .ascii "-"
.section .text
.globl _start
_start:
   nop
   leal string1, %edi
   leal string2, %esi
   movl length, %ecx
   lodsb
   cld
   repne scasb
   jne notfound
   subw length, %cx
   neg %cx
   movl $1, %eax
   movl %ecx, %ebx
   int $0x80
notfound:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

scastest1.s程序把要扫描的字符串的内存位置加载到EDI寄存器中，使用LODSB指令把要搜索的字符加载到AL寄存器中，然后把字符串的长度存放到ECX寄存器中。所有这些工作完成之后，使用REPNE SCASB指令扫描字符串，获得搜索字符的位置。如果没有找到这个字符，JNE指令就会跳转到标签notfound的分支。如果找到了这个字符，那么现在它距离字符串末尾的位置就存放在ECX寄存器中。从ECX的值中减去字符串的长度，然后使用NEG指令改变结果值的符号，以便生成找到的字符在字符串中的位置。这个位置被加载到EBX寄存器中，使它成为程序终止之后的结果代码:

```
$ ./scastest1
$ echo $?
16
$
```

输出显示在字符串的第16个位置找到了“-”字符。

### 10.4.2 搜索多个字符

虽然SCASW和SCASL指令可以用于搜索2个或者4个字符的序列，但是使用它们的时候必须要谨慎。它们可能不像读者期望的那样执行。

SCASW和SCASL指令扫描字符串，查找AX或者EAX寄存器中的字符序列，但是它们并不

进行逐字符的比较。而是每次比较之后，EDI寄存器要么递增2（对于SCASW），要么递增4（对于SCASL），而不是递增1。这就是说字符序列也必须按照适当的顺序出现在字符串中。scastest2.s程序演示这个问题：

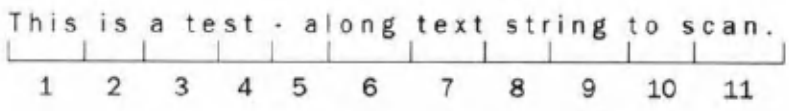
```
# scastest2.s - An example of incorrectly using the SCAS instruction
.section .data
string1:
   .ascii "This is a test - a long text string to scan."
length:
   .int 11
string2:
   .ascii "test"
.section .text
.globl _start
_start:
   nop
   leal string1, %edi
   leal string2, %esi
   movl length, %ecx
   lodsl
   cld
   repne scasl
   jne notfound
   subw length, %cx
   neg %cx
   movl $1, %eax
   movl %ecx, %ebx
   int $0x80
notfound:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

scastest2.s程序试图在字符串中查找字符序列“test”。它把整个搜索字符串加载到EAX寄存器中，然后使用SCASL指令一次检查字符串的4个字节。注意ECX寄存器没有被设置为字符串的长度，而是被设置为REPNE指令遍历整个字符串所需的迭代次数。因为每次迭代检查4个字节，所以ECX寄存器的值是整个字符串长度44的四分之一。

如果运行程序并且查看结果代码，会发现不期望看到的情况：

```
$ ./scastest2
$ echo $?
0
$
```

SCASL指令在字符串中没有找到字符序列“test”。显然，出现了某些错误。问题在于REPNE指令执行迭代的方式。图10-2演示问题是如何发生的。




REPNE SCASL迭代

图 10-2

REPNE指令的第一次迭代比较4个字节的“This”和EAX中的字符序列。因为它们不匹配，所以ECX寄存器递增4，然后检查下面的4个字节“is”。可以从图10-2看出，被测试的每一组4个字节都不和搜索字符序列相匹配，尽管这个序列确实在这个字符串中。

虽然SCASW和SCASL指令不能很好地处理字符串，但是，在数据数组中搜索非字符串数据时，它们还是有用的。SCASW指令可以用于2字节数组，SCASL指令可以用于处理4字节数组。

### 10.4.3 计算字符串长度

SCAS指令的一个非常有用的功能是确定零结尾（也称为空结尾）的字符串的长度。这些字符串经常在C程序中使用，但是也通过使用.asciz声明在汇编语言程序中使用。对于零结尾的字符串，要搜索的显然是零的位置，并且计数找到零经过了多少个字符。strsize.s程序演示这种操作：

```
# strsize.s - Finding the size of a string using the SCAS instruction
.section .data
string1:
   .asciz "Testing, one, two, three, testing.\n"
.section .text
.globl _start
_start:
   nop
   leal string1, %edi
   movl $0xffff, %ecx
   movb $0, %al
   cld
   repne scasb
   jne notfound
   subw $0xffff, %cx
   neg %cx
   dec %cx
   movl $1, %eax
   movl %ecx, %ebx
   int $0x80
notfound:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

strsize.s程序把要测试的字符串的内存位置加载到EDI寄存器中，把假设的字符串长度加载到ECX寄存器中。字符串长度值0xffff表明这个程序只能用于长度最大为65 535的字符串。ECX寄存器将跟踪在字符串中找到表示结尾的零经过了多少次迭代。如果SCASB指令找到了零，就必须通过ECX寄存器的值计算它的位置。从ECX寄存器的值中减去它的初始值，并且改变结果的符号，就完成了这个工作。因为这个长度包含表示结尾的零，所以最终值必须减1才能显示字符串真正的长度。计算的结果存放在EBX寄存器中，以便可以通过查看程序的结果代码获得它：

```
$ ./strsize
$ echo $?
35
$
```

## 10.5 小结

字符串是处理必须和人类进行交互的程序时的重要部分。创建、传送、比较和扫描字符串是汇编语言程序中必须执行的极为重要的功能。本章介绍IA-32的字符串指令，并且演示如何在汇编语言程序中使用它们。

MOVS指令系列提供把字符串从一个内存位置传送到另一个内存位置的方法。这些指令把ESI寄存器引用的字符串传送到EDI寄存器引用的位置。MOVS指令有3种格式，MOVSB（用于字节）、MOVSW（用于字）和MOVSL（用于双字），它们提供在内存中传送字符串的快捷方法。为了传送更多的内存，REP指令允许按照预先设置的次数重复执行MOVS指令。ECX寄存器用作确定MOVS指令将执行多少次迭代的计数器。要记住，ECX寄存器计数的是迭代次数，而不一定是字符串长度，这一点很重要。ESI和EDI寄存器按照传送的数据的长度自动递增或者递减。EFLAGS寄存器的DF标志用于控制寄存器改变的方向。

LODS指令系列用于把字符串值加载到EAX寄存器中。LODSB指令把一个字节传送到AL寄存器中，LODSW指令把一个字传送到AX寄存器中，LODSL指令把一个双字传送到EAX寄存器中。字符串值被存放到寄存器中之后，可以对它执行很多不同的功能，比如把ASCII值从小写字母改变为大写字母。执行操作之后，可以使用STOS指令把字符串的字符传送回内存中。STOS指令包括STOSB（存储字节）、STOSW（存储字）和STOSL（存储双字）。

对字符串执行的一个非常方便的功能是比较不同字符串值的能力，这是通过CMPS指令实现的。要比较的字符串的内存位置被加载到ESI和EDI寄存器中。CMPS指令的结果设置在EFLAGS寄存器中，使用通常的标志。这使得可以使用标准的条件分支指令检查EFLAGS标志的设置并且相应地跳转到分支。REPE指令可以用于比较更长的字符串，它比较每个单独字符，直到两个字符不相同为止。要比较的字符串的长度设置在ECX寄存器中。同样，EFLAGS寄存器可以用于确定两个字符串是否相同。

除了比较字符串之外，扫描字符串搜索特定的字符或者字符序列通常是有用的。SCAS指令用于扫描字符串搜索存储在EAX寄存器中的字符。SCAS指令经常和REPNE指令一起使用，以便在整个字符串的长度内重复扫描功能。扫描将持续到找到搜索字符，或者到达字符串的末尾为止。要扫描的字符串的长度存放在ECX寄存器中，结果设置在EFLAGS寄存器中。

使用这些专门的字符串功能，还有前面章节介绍的专门的数学功能，可以开始构建汇编语言函数库，以便在所有程序中使用。第11章“使用函数”演示如何把代码功能分隔到自包含的库中，在汇编程序中需要这些库的时候可以利用它们。这样，由于不必在所有程序中重复编写相同的函数，能够节省很多时间。

# 第11章 使 用 函 数

当开始为不同项目编写各种类型的实用程序之后，你将意识到在同一程序中多次编写相同的实用程序，或者可能在不同的程序中使用相同的实用程序。不必在每次需要时都重复编写实用程序代码，可以创建独立的函数，从而在需要该实用程序的任何时候调用它。

本章介绍函数是什么，以及如何在程序中使用它们。读者将学习如何在汇编语言中使用函数，包括如何创建函数以及如何在汇编语言程序中使用它们的例子。之后，将学习如何使用C调用的约定创建汇编语言函数，以及如何把汇编语言函数分隔为独立于主程序的文件。最后，因为传递参数对于函数和对于程序都是重要的，所以读者将学习如何从汇编语言程序中读取和处理命令行参数，并且研究一个如何在应用程序中使用命令行参数的例子。

## 11.1 定义函数

通常，应用程序需要对不同的数据集合多次执行相同的过程或者处理。当应用程序中需要相同的代码时，不必多次重新编写代码，有时候最好创建包含代码的单一函数（function），然后可以在程序中的任何位置调用这个函数。函数包含完成特定例程所需的所有代码，而且不需要主程序中任何代码的帮助。数据从主程序传递给函数，然后结果返回给主程序。

调用函数时，程序的执行路径被改变，切换到函数代码中的第一条指令。处理器从这个位置开始执行指令，直到函数表明它可以把控制返回到主程序中的原始位置。图11-1演示这种情况。

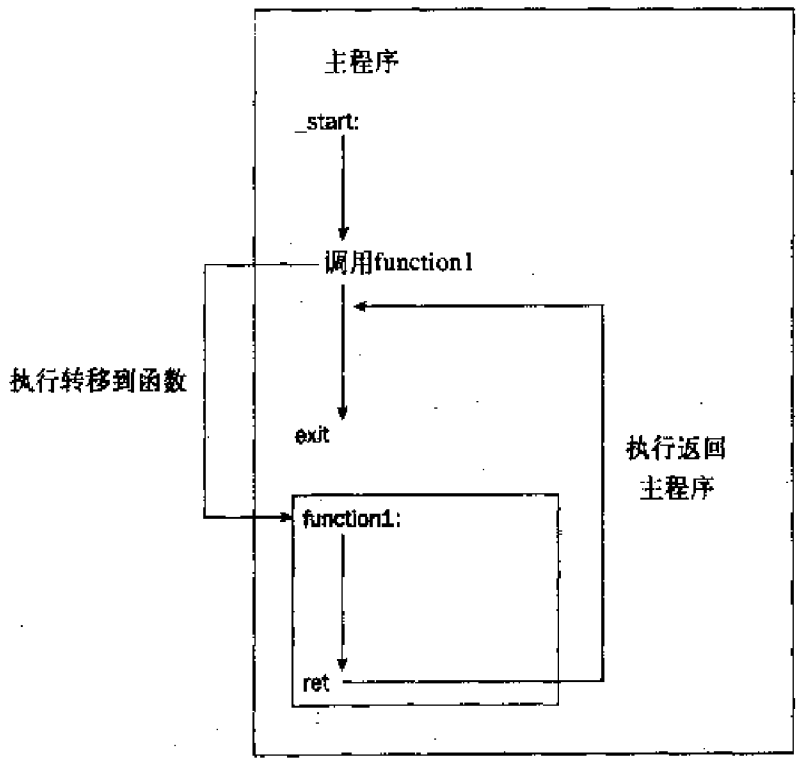



图 11-1

大多数高级语言提供了在程序中编写和使用函数的方法。函数可以和主程序包含在相同的源代码文件中，并且一起编译，函数也可以位于独立的源代码文件中，并且连接到主程序。

ctest.c程序演示一个简单的C语言函数，它计算给定半径的圆的面积：

```
/* ctest.c - An example of using functions in C */
#include <stdio.h>

float function1(int radius)
{
   float result = radius * radius * 3.14159;
   return result;
}

int main()
{
   int i;
   float result;
   i = 10;
   result = function1(i);
   printf("Radius: %d,  Area: %f\n", i, result);
   i = 2;
   result = function1(i);
   printf("Radius: %d,  Area: %f\n", i, result);

   i = 120;
   result = function1(i);
   printf("Radius: %d,  Area: %f\n", i, result);
   return 0;
}
```

这个函数定义了必须的输入值，还有函数将生成的输出的类型：

```
float function1(int radius)
```

这个函数的定义显示输入值是整数，函数生成的输出是浮点值。在程序中调用函数时，必须使用相同的格式：

```
result = function1(i);
```

函数调用的内部放置的变量被定义为整数，函数的输出存放在变量result中，它被定义为浮点值。为了编译这个程序，可以使用GNU的C编译器（gcc）：

```
$ gcc -o ctest ctest.c
$./ctest
Radius: 10,  Area: 314.158997
Radius: 2,  Area: 12.566360
Radius: 120,  Area: 45238.894531
$
```

既然读者已经看到了函数如何工作的一般情况，现在该使用汇编语言创建一些函数了。

## 11.2 汇编函数

在汇编语言中编写函数和使用C语言编写它们的方式类似。为了创建函数，必须定义它的输入值、它如何处理输入值以及它的输出值。函数定义好之后，就把它编写到汇编语言应用程序中。

本节介绍如何在汇编语言程序中创建函数，并且提供在程序中使用函数的简单例子。

### 11.2.1 编写函数

在汇编语言程序中创建函数需要3个步骤：

1) 定义需要的输入值。

2) 定义对输入值执行的操作。

3) 定义如何生成输出值以及如何把输出值传递给发出调用的程序。

本节逐步讲解这些步骤，使用求圆的面积的函数作为例子。

1. 定义输入值

很多函数都需要某种形式的输入数据。程序员必须定义程序如何把这些信息传递给函数。基本上，可以使用3种技术：

- 使用寄存器
- 使用全局变量
- 使用堆栈

本节中的简单例子显示使用寄存器技术把输入值传递给函数。其他两种技术（全局变量和堆栈）将在本章后面详细讨论。

当主程序调用函数时，主程序在调用函数时停止，函数从这个位置开始执行。函数可以访问位于内存中和寄存器中的任何数据。使用寄存器把输入值传递给函数是快速而且方便的。

*使用寄存器在主程序和函数之间传递数据时，记住使用数据的正确数据类型。数据存放在寄存器中时使用的数据类型必须是函数期待的相同类型。*

2. 定义函数处理

函数被编写为程序中的一般汇编语言代码。在源代码文件中，函数指令必须和主程序的其余指令分离开。

函数有别于主程序的其余部分的地方是为汇编器定义函数的方式。不同汇编器使用不同的方法定义函数。

为了在GNU汇编器中定义函数，必须在程序中把函数名称声明为标签。为汇编器声明函数名称，可以使用.type命令：

```
.type func1, @function
func1:
```

.type命令通知GNU汇编器，func1标签定义将在汇编语言程序中使用的函数的开始。func1标签定义函数的开始。func1标签后面的第一条指令是函数的开头。

在函数中，可以使用主程序中那样的代码。可以访问寄存器和内存位置，可以使用专门特性，比如FPU、MMX和SSE。

函数的结束由RET指令定义。执行到RET指令时，程序控制返回主程序，返回的位置是紧跟在调用函数的CALL指令后面的指令。

3. 定义输出值

当函数完成对数据的处理时，很可能希望把结果传递回发出调用的程序区域。函数必须能

够把结果传递回主程序，以便主程序能够利用此数据做进一步的处理或者显示。和输入值的技术类似，有多种方式完成传送结果的工作，但是下面两种是最常见的：

- 把结果存放在一个或者多个寄存器中。
- 把结果存放在全局变量内存位置中。

这里使用的函数例子使用单一寄存器把结果传递回主程序。在本章后面，将学习使用全局变量技术把数据传递回主程序。

4. 创建函数

既然读者已经了解了创建汇编语言函数的基础知识，现在该创建一个函数了。这个代码片断显示如何在汇编语言程序中创建一个函数area，它执行计算圆面积的功能：

```
.type area, @function
area:
   fldpi
   imull %ebx, %ebx
   movl %ebx, value
   fildS value
   fmulp %st(0), %st(1)

   ret
```

函数area使用FPU，根据半径计算圆形的面积。输入值（半径）被假设为整数值，当调用函数时输入值位于EBX寄存器中。

使用FLDPI指令把计算需要的pi值加载到FPU寄存器堆栈中。接下来，求EBX寄存器中的值的平方，然后存储在主程序已经在内存中定义的内存位置中（记住函数可以自由访问主程序定义的内存位置）。

使用FILDS指令把半径的平方值加载到FPU堆栈的顶部，把pi值移动到ST(1)的位置。下面，使用FMULP指令使第一个和第二个FPU堆栈位置相乘，把结果存放在ST(1)的位置，然后把ST(0)值弹出堆栈，把最终结果留在ST(0)寄存器中。

对使用函数area的程序的要求如下：

- 输入值必须作为整数值存放在EBX寄存器中。
- 必须在主程序中创建名为value的4字节内存位置。
- 输出值被放在FPU的ST(0)寄存器中。

在主程序中使用函数area时遵守这些要求是至关重要的。例如，如果把半径值作为浮点数据类型存放在EBX寄存器中，那么整个面积计算就会是错误的。

为了使应用程序能够使用函数，这里所示的area函数例子必须遵守一组复杂的要求。如果处理使用许多函数的大型应用程序，那么跟踪哪个函数具有哪些需求是很困难的。对此有弥补的方式，这在11.3节中介绍。

### 11.2.2 访问函数

创建好函数之后，就可以在程序中的任何位置访问它。CALL指令用于把控制从主程序传递到函数。CALL指令使用单一操作数：

```
call function
```

其中function是要调用的函数的名称。记住在执行CALL指令之前，要把所有输入值放在正确的位置中。

functest1.s程序演示在程序中多次调用area函数：

```
# functest1.s - An example of useing functions
.section .data
precision:
   .byte 0x7f, 0x00
.section .bss
   .lcomm value, 4
.section .text
.globl _start
_start:
   nop
   finit
   fldcw precision

   movl $10, %ebx
   call area

   movl $2, %ebx
   call area

   movl $120, %ebx
   call area

   movl $1, %eax
   movl $0, %ebx
   int $0x80

.type area, @function
area:
   fldpi
   imull %ebx, %ebx
   movl %ebx, value
   filds value
   fmulp %st(0), %st(1)

   ret
```

函数area就是前面给出的代码片断。主程序首先初始化FPU并且将它设置为使用单精度浮点结果（参见第9章）。下面，三次调用area函数，每次在EBX寄存器中使用不同的值。函数的输出作为单精度浮点数据类型存放在EAX寄存器中。这个例子中没有显示结果。可以在运行程序时从调试器查看结果。

汇编和连接程序之后，在调试器中运行它，并且查看程序如何执行。当执行CALL指令时，执行的下一条指令是函数中的第一条指令：

```
$ gdb -q functest1
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file functest1.s, line 11.
(gdb) run
Starting program: /home/rich/palp/chap11/functest1

Breakpoint 1, _start () at functest1.s:11
```

```
11          finit
Current language:  auto; currently asm
(gdb) s
12          fldcw precision
(gdb) s
14          movl $10, %ebx
(gdb) s
15          call area
(gdb) s
29          fldpi
(gdb)
```

继续单步执行程序，执行函数中的下一条指令，一直执行到RET指令，它返回主程序：

```
(gdb) s
37          ret
(gdb) s
17          movl $2, %ebx
(gdb)
```

返回到主程序之后，可以查看EAX寄存器中作为浮点值的结果值：

```
(gdb) print/f $eax
$1 = 314.159271
(gdb)
```

函数就像我们期望的那样执行，从第一个半径值（10）生成正确的面积值。这一处理继续对其他值和函数调用执行。

### 11.2.3　函数的放置

读者也许会注意到在functest1.s程序中，函数代码被安排在主程序的源代码的后面。也可以把函数代码放在主程序的源代码的前面。在第4章“汇编语言程序范例”中讲过，当连接源代码目标文件时，连接器查找标签为_start的代码段作为要执行的第一条指令。可以把任意数量的函数代码放在_start之前，而且不会影响主程序的开始。

另外，就像functest1.s程序中演示过的，和某些高级语言不同，不必在主程序中调用函数之前定义函数。所有CALL指令都会查找定义函数开始的标签来获得指令指针。

### 11.2.4　使用寄存器

虽然area函数在处理过程中只使用单一输出寄存器，但是更加复杂的函数也许并不如此。函数经常使用寄存器处理数据。不能保证函数完成时的寄存器状态和函数被调用之前的状态相同。

从程序调用函数时，程序员应该知道函数使用哪些寄存器进行它的内部处理。当执行返回主程序时，函数中使用的任何寄存器（还有内存位置）也许是，也许不是原来的值。

如果被调用的函数修改主程序使用的寄存器，那么在调用函数之前保存寄存器的当前状态，并且在函数返回之后恢复寄存器的状态，这是至关重要的。在调用函数之前，可以使用PUSH指令单独地保存特定寄存器，也可以使用PUSHA指令同时保存所有寄存器。类似地，可以使用POP指令单独地恢复寄存器的原始状态，也可以使用POPA指令同时恢复所有寄存器的状态。

在函数调用之后恢复寄存器值时要谨慎。如果函数返回的值存放在寄存器中，那么在恢复原始寄存器值之前，必须把它传送到安全位置。

### 11.2.5 使用全局数据

读者已经在area函数例子中看到，函数可以访问主程序中定义的内存位置。因为程序中的所有函数都可以访问这些内存位置，所以它们称为全局变量（global variable）。函数可以将全局变量用于任何目的，包括在主程序和函数之间传递数据。

functest2.s程序演示使用全局变量在函数和主程序之间传递输入值和输出值：

```
# functest2.s - An example of using global variables in functions
.section .data
precision:
   .byte 0x7f, 0x00
.section .bss
   .lcomm radius, 4
   .lcomm result, 4
   .lcomm trash, 4
.section .text
.globl _start
_start:
   nop
   finit
   fldcw precision

   movl $10, radius
   call area

   movl $2, radius
   call area

   movl $120, radius
   call area

   movl $1, %eax
   movl $0, %ebx
   int $0x80

.type area, @function
area:
   fldpi
   filds radius
   fmul %st(0), %st(0)
   fmulp %st(0), %st(1)
   fstps result

   ret
```

functest2.s程序修改了area函数，使它不需要使用任何通用寄存器。而是函数从全局变量radius获得输入值，把它加载到FPU中，在FPU中执行所有数学操作，然后把FPU的ST(0)寄存器弹出到一个全局变量中，主程序可以访问这个全局变量。因为不使用寄存器保存输入值，所以主程序必须把每个输入值加载到全局变量radius中，而不是寄存器中。

汇编和连接程序之后，可以在调试器中运行它，在程序处理过程中查看全局变量。下面是第一次调用area函数之后的值：

```
(gdb) x/d &radius
0x80490d4 <radius>:     10
(gdb) x/f &result
0x80490d8 <result>:     314.159271
(gdb)
```

结果正确地存放在了result内存位置中。第一次调用函数之后，新的值被加载到内存位置radius中，再次调用函数。这样继续到第三次调用。

使用全局内存位置传递参数和返回结果不是常见的程序设计方法，即使在C和C++程序设计中也不是。下一节介绍更加常见的在函数之间传递值的方法。

## 11.3　按照C样式传递数据值

读者会发现，在函数中处理输入值和输出值可用的选择有很多。虽然这看上去像是好事情，但实际上这可能造成问题。如果为大型项目编写函数，确保正确使用每个函数所需的文档就是无法计数的。试图跟踪哪个函数使用哪些寄存器和全局变量，或者使用哪些寄存器和全局变量传递哪些参数，会是程序员的恶梦。

为了帮助解决这个问题，必须使用某一标准一致地存放输入参数以便函数获取，并且一致地存放输出值便于主程序获取。为IA-32平台创建代码时，在从C函数编译出来的汇编语言代码中，大多数C编译器使用标准方法处理输入和输出值。这种方法也适用于任何汇编语言程序，即使它们不是来源于C程序。

C的把输入值传递给函数的解决方案是使用堆栈。主程序可以访问堆栈，程序中使用的任何函数也可以。这样就创建了在通用的位置在主程序和函数之间传递数据的明确途径，而无需担心破坏寄存器或者定义全局变量。

同样，C样式定义了把值返回主程序的通用方法——EAX寄存器用于32位结果（比如短整数），EDX：EAX寄存器对用于64位整数值，FPU的ST(0)寄存器用于浮点值。

下面几节回顾堆栈如何工作，以及如何使用它把输入值传递给函数。

### 11.3.1　回顾堆栈

第5章简要地讨论过堆栈的基本操作。堆栈由为程序分配的内存的末尾处保留的内存位置构成。ESP寄存器用于指向内存中堆栈的顶部。只能把数据存放到堆栈的顶部，并且只能从堆栈的顶部删除数据。

通常使用PUSH指令把数据存放到堆栈中。这把数据存放在为堆栈保留的内存区域的底部，并且把堆栈指针（ESP寄存器）中的值递减为新的数据位置。

为了从堆栈获得数据，使用POP指令。这把数据传送到寄存器或者内存位置中，并且把ESP寄存器的值递增为指向前一个堆栈数据值。

### 11.3.2　在堆栈之中传递函数参数

在调用函数之前，主程序把函数所需的输入参数存放到堆栈的顶部。当然，程序员必须知

道函数希望数据值存放在堆栈中的顺序，但这通常是函数文档的一部分。C样式要求参数存放到堆栈中的顺序和函数的原型中的顺序相反。

执行CALL指令时，它把发出调用的程序的返回地址也存放到堆栈的顶部，这样函数可以知道返回到什么位置。这使堆栈的状态如图11-2所示。

堆栈指针（ESP）指向堆栈的顶部，这里加载了返回地址。在堆栈中，函数的所有输入参数都位于返回地址的“下面”。把值弹出堆栈以获得输入参数会导致一个问题，因为返回地址可能在处理过程中丢失。替换的做法是，使用其他的方法从堆栈获得输入参数。

第5章讨论过使用寄存器进行间接寻址的主题。这种技术提供根据寄存器中的变址值访问内存中的位置的方法。因为ESP指针指向堆栈的顶部（这里包含函数的返回地址），函数可以根据ESP寄存器使用间接寻址的方式访问输入参数，不必把值弹出堆栈。图11-3演示这种情况。

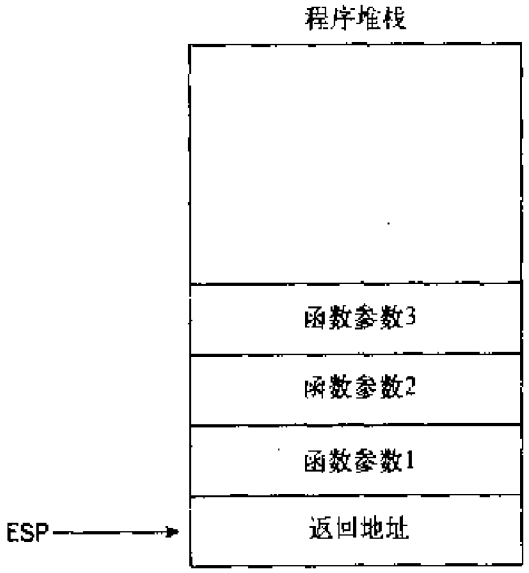



图 11-2

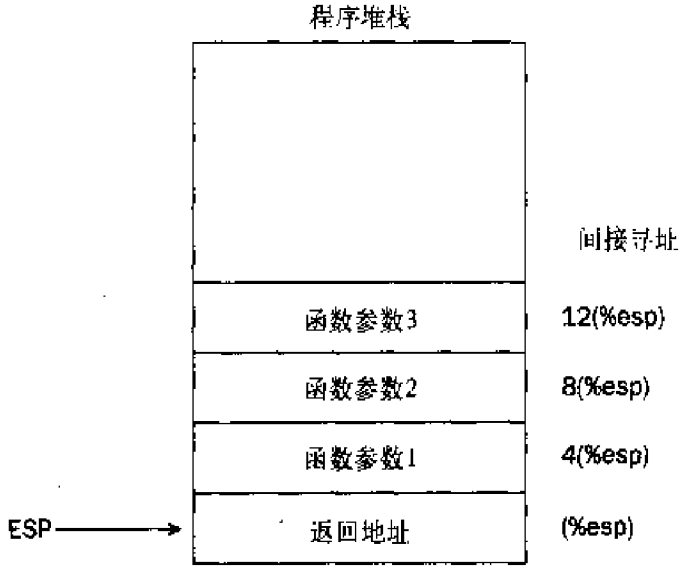



图 11-3

可以通过距离ESP寄存器值的偏移量使用间接寻址访问每个参数，而不必使用POP指令把值弹出堆栈。

但是，这种技术有个问题。因为在函数中，函数处理的某个部分可能包含把数据压入堆栈的操作。如果发生这种情况，就会改变ESP堆栈指针的位置，并且丢失用于访问堆栈中的参数的间接寻址值。

为了避免这个问题，通用的做法是进入函数时把ESP寄存器复制到EBP寄存器。这样确保有一个寄存器永远包含指向调用函数时的堆栈顶部的正确指针。函数执行过程中压入堆栈的任何数据都不会影响EBP寄存器的值。为了避免破坏原始的EBP寄存器值，如果主程序中使用它的话，在复制ESP寄存器的值之前，

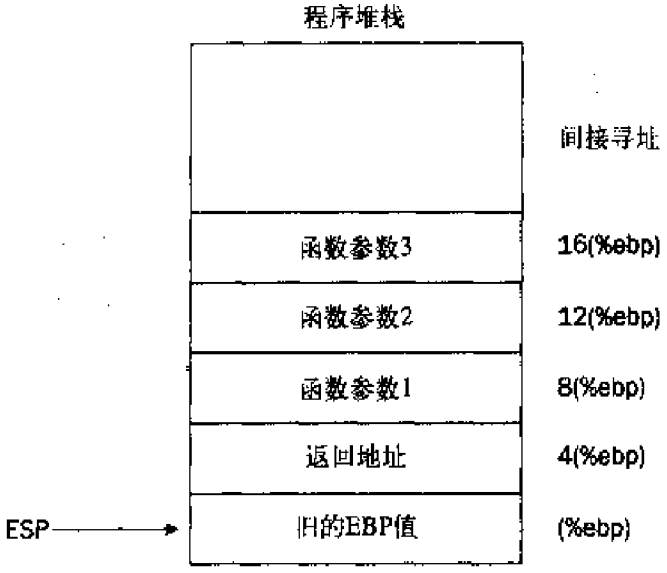



图 11-4

EBP寄存器的值也被存放到堆栈中。这样就产生了图11-4所示的情况。

现在EBP寄存器包含堆栈的开始位置（现在是旧的EBP寄存器值）。来自主程序的第一个输入参数位于间接寻址位置8（%ebp），第二个参数位于12（%ebp）的位置，等等。可以在函数中使用这些值，而无需担心其他值被压入堆栈或者从堆栈删除。

### 11.3.3 函数开头和结尾

使用堆栈引用函数的输入数据的技术创建了一组标准指令，使用C函数样式技术编写的所有函数都使用它们。下面这个代码片断演示在函数代码的开头和结尾使用什么指令：

```
function:
   pushl %ebp
   movl %esp, %ebp
   .
   .
   movl %ebp, %esp
   popl %ebp
   ret
```

函数代码开头的前两条指令把EBP的原始值保存到堆栈的顶部，然后把当前ESP堆栈指针（现在指向堆栈中EBP的原始值）复制到EBP寄存器。

函数处理完成之后，函数的最后两条指令获取存储在EBP寄存器中的原始的ESP寄存器值，并且恢复EBP寄存器的原始值。重新设置ESP寄存器的值确保当执行返回主程序时，函数中存放到堆栈中，但是还没有清除的任何数据都会被丢弃（否则，RET指令就会返回到错误的内存位置）。

ENTER和LEAVE指令被设计为专门用于建立函数开头（ENTER指令）和结尾（LEAVE指令）。可以使用它们替代手工地创建开头和结尾。

### 11.3.4 定义局部函数数据

当程序控制权在函数代码中时，处理过程很可能需要在某个位置存储数据元素。前面讨论过，可以在函数代码中使用寄存器，但是这种方式只提供数量有限的工作区域。也可以使用全局变量来处理数据，但是问题在于这会额外要求主程序为函数提供专门的数据元素。当在函数中为数据元素的存储寻找方便的位置时，堆栈再一次提供了帮助。

EBP寄存器被设置为指向堆栈的顶部之后，函数中使用的任何附加的数据都可以存放在堆栈中这个指针之后，这不会影响对输入值的访问。如图11-5所示。

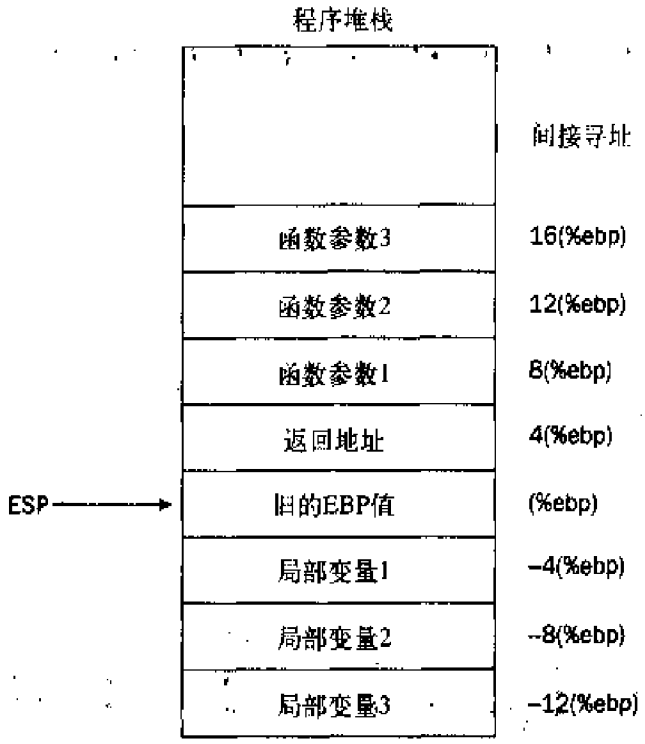



图 11-5

在堆栈中定义局部变量之后，可以使用EBP寄存器很容易地引用它们。假设对于4字

节的数据值，可以通过引用-4（%ebp）访问第一个局部变量，引用-8（%ebp）访问第二个局部变量。

这种设置还有一个残留的问题。如果函数把任何数据压入堆栈，ESP寄存器仍然指向局部变量被存放之前的位置，并且将覆盖这些变量。

为了解决这个问题，在函数代码的开始添加了另一行，通过从ESP寄存器减去一个值，为局部变量保留一定数量的堆栈空间。图11-6显示堆栈中的情况。

现在，如果把任何数据压入堆栈，数据会被存放在局部变量下面，这保护了它们，使得仍然可以通过EBP寄存器指针访问它们。一般的ESP寄存器仍然可以用于把数据压入堆栈和弹出堆栈，且不会影响局部变量。

到达函数的结尾并且ESP寄存器被设置回其原始值时，局部变量会从堆栈丢失，并且无法从发出调用的程序使用ESP或者EBP寄存器直接访问它们（这就是“局部变量”这个术语的由来）。

现在函数开头代码必须包含附加的一行，它通过向下移动堆栈指针，为局部变量保留空间。必须记住为函数中所需的所有局部变量保留足够的空间。新的开头就像下面这样：

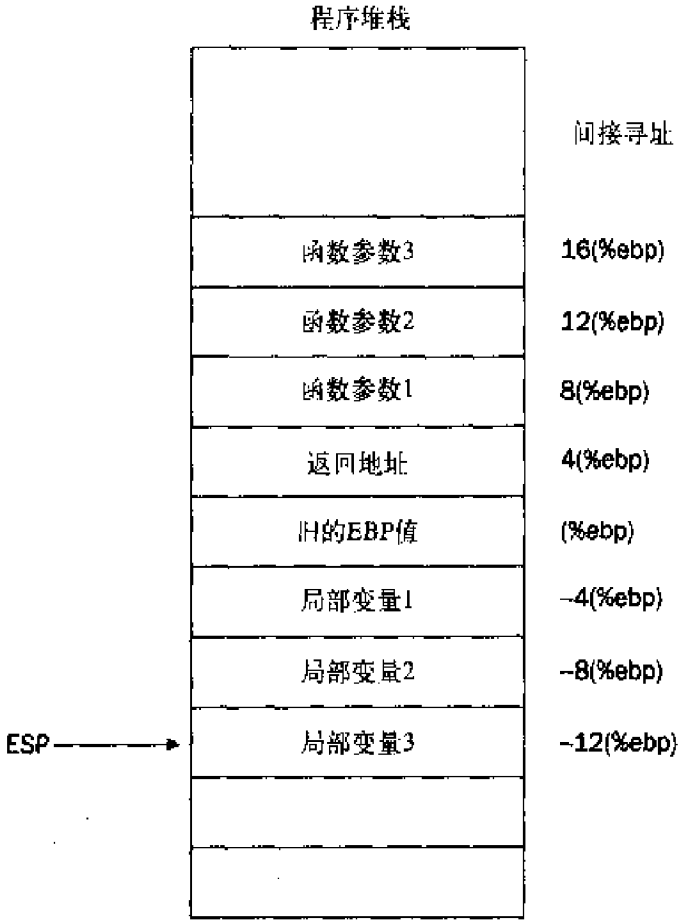



图 11-6

```
function:
   pushl %ebp
   movl %ebp, %esp
   subl $8, %esp
   .
   .
```

这些代码保留8字节供局部变量使用。这些空间可以用作4个字值、或者2个双字值。

### 11.3.5 清空堆栈

当使用C样式的函数调用时，还有一个细节需要考虑。调用函数之前，发出调用的函数把所有必须的输入值存放到堆栈中。函数返回时，这些值仍然在堆栈中（因为函数访问它们且不把它们弹出堆栈）。如果主程序使用堆栈进行其他操作，它很可能希望从堆栈中删除旧的输入值，以便使堆栈恢复到函数调用之前的状态。

虽然可以使用POP指令完成这个工作，但是也可以把ESP堆栈指针移动回函数调用之前的原始位置。使用ADD指令把压入堆栈的数据元素的长度加上去，就完成了这个工作。

例如，如果把两个4字节的整数值存放到堆栈中，然后调用函数，那么就必须使ESP寄存器

加上8以便把数据清除出堆栈:

```
pushl %eax
pushl %ebx
call compute
addl $8, %esp
```

这样确保堆栈恢复到应该的状态，以便主程序的其余部分使用。

### 11.3.6 范例

既然读者已经了解了如何使用堆栈把输入值传递给函数，以及如何使用堆栈存储函数中的局部变量，现在是研究例子的时候了。functest3.s程序演示这些组件如何在一起工作简化area函数例子:

```
# functest3.s - An example of using C style functions
.section .data
precision:
   .byte 0x7f, 0x00
.section .bss
   .lcomm result, 4
.section .text
.globl _start
_start:
   nop
   finit
   fldcw precision

   pushl $10
   call area
   addl $4, %esp
   movl %eax, result

   pushl $2
   call area
   addl $4, %esp
   movl %eax, result

   pushl $120
   call area
   addl $4, %esp
   movl %eax, result

   movl $1, %eax
   movl $0, %ebx
   int $0x80

.type area, @function
area:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   fldpi
   filds 8(%ebp)
   fmul %st(0), %st(0)
   fmulp %st(0), %st(1)
```

```
fstps -4(%ebp)
movl -4(%ebp), %eax

movl %ebp, %esp
popl %ebp
ret
```

area函数代码被修改为使用堆栈获得半径输入值，并且使用一个局部函数变量把结果作为单精度浮点值从ST(0)寄存器传送到EAX寄存器。函数的结果返回到EAX寄存器中，而不是ST(0)寄存器中，这帮助演示使用局部变量。在第14章中将看到C程序希望结果位于ST(0)寄存器中。

在标准函数开头之后（包括在堆栈中保留4字节用于局部变量），pi值和半径值被加载到了FPU堆栈中：

```
fdpi
filds 8(%ebp)
```

8(%ebp)值引用发出调用的程序存放在堆栈中的第一个参数。计算出面积结果之后，它被存放到局部变量数据中：

```
fstps -4(%ebp)
```

-4(%ebp)值引用堆栈中的第一个局部变量。这个值被存放在EAX寄存器中，主程序从这里获得它。

之后，使用标准函数结尾指令替换ESP和EBP指针值，并且把控制返回主程序。

注意主程序中用于每个半径值的新代码：

```
pushl $10
call area
addl $4, %esp
movl %eax, result
```

在调用area函数之前，每个半径值都被压入到堆栈中。函数完成时，通过使ESP寄存器加上4（半径值的数据长度），堆栈被复位到它的先前位置。然后把结果从EAX寄存器传送到全局变量内存位置。

### 11.3.7 在操作之中监视堆栈

通过使用调试器，可以监视程序运行时所有数据值是如何存放到堆栈区域中的。汇编和连接程序之后，在调试器中运行程序，在程序的开始设置断点，并且查看堆栈指针设置在什么位置：

```
$ gdb -q functest3
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file functest3.s, line 11.
(gdb) run
Starting program: /home/rich/palp/chap11/functest3

Breakpoint 1, _start () at functest3.s:11
11          finit
Current language:  auto; currently asm
(gdb) print $esp
$1 = (void *) 0xbffff950
(gdb)
```

当前堆栈指针被设置为指向内存地址0xbffff950。单步执行程序，直到CALL指令前面的第一条PUSHL指令，并且查看堆栈指针值和最后堆栈位置中的数据元素的值：

```
(gdb) s
14          pushl $10
(gdb) s
15          call area
(gdb) print $esp
$2 = (void *) 0xbffff94c
(gdb) x/d 0xbffff94c
0xbffff94c:     10
(gdb)
```

执行PUSHL指令之后，新的堆栈指针值比原始值小4字节（正如我们期望的）。现在压入到堆栈中的数据值显示在这个内存位置。现在，单步执行CALL指令，再次查看堆栈：

```
(gdb) s
35          pushl %ebp
(gdb) print $esp
$3 = (void *) 0xbffff948
(gdb) x/x 0xbffff948
0xbffff948:     0x08048085
(gdb) x/d 0xbffff94c
0xbffff94c:     10
(gdb)
```

ESP寄存器的值再一次递减，存储在这里的值是紧跟在CALL指令后面的指令的内存位置。输入参数仍然位于下一个堆栈位置中。

现在程序的控制在area函数中。单步执行PUSHL指令之后，ESP寄存器再次递减，EBP的原始值被存放到堆栈中：

```
(gdb) s
36          movl %esp, %ebp
(gdb) print $esp
$5 = (void *) 0xbffff944
 (gdb) x/x 0xbffff944
0xbffff944:     0x00000000
(gdb)
```

单步执行把ESP寄存器的值加载到EBP中的指令，现在可以使用EBP进行间接寻址来访问堆栈中的数据了。使用调试器，能看到3个值存储在堆栈中：

```
(gdb) x/3x ($ebp)
0xbffff944:     0x00000000      0x08048085      0x0000000a
(gdb)
```

现在堆栈包含EBP的原始值、来自CALL指令的返回地址和使用PUSH指令存放到堆栈中的输入值。

下一步，减小ESP寄存器，为局部变量留出空间：

```
(gdb) s
area () at functest3.s:37
37          subl $4, %esp
```

```
(gdb) s
38          fldpi
(gdb) print $esp
$1 = (void *) 0xbffff940
(gdb)
```

新的ESP位置是0xbffff940，EBP值是0xbffff944。这样为存储32位单精度浮点值留出了4字节。pi值以及输入参数被加载到FPU堆栈中，可以在调试器中使用info all命令查看它们：

```
(gdb) info all
.
.
st0   10      (raw 0x4002a000000000000000)
st1   3.14159265358979323851280895940618620 (raw 0x4000c90fdaa22168c235)
(gdb)
```

使用FILDS指令，输入参数被成功地加载到了FPU堆栈中。

单步执行area函数的其余指令，可以在把结果存放到堆栈中的局部变量时停下，并且查看情况如何：

```
(gdb) s
42          fstps -4(%ebp)
(gdb) s
43          movl -4(%ebp), %eax
(gdb) x/4x 0xbffff940
0xbffff940:     0x439d1463      0x00000000      0x08048085      0x0000000a
(gdb) x/f 0xbffff940
0xbffff940:     314.159271
(gdb)
```

结果值被存放到堆栈中的-4（%ebp）位置。在调试器中，通过使用x/f读取内存位置并且把输出设置为浮点格式，可以查看这一情况。

函数中的最后几个步骤把ESP和EBP寄存器恢复到它们的原始值：

```
(gdb) s
45          movl %ebp, %esp
(gdb) s
46          popl %ebp
(gdb) print $esp
$2 = (void *) 0xbffff944
(gdb) print $ebp
$3 = (void *) 0xbffff944
(gdb) s
area () at functest3.s:47
47          ret
(gdb) print $esp
$4 = (void *) 0xbffff948
(gdb) print $ebp
$5 = (void *) 0x0
(gdb)
```

现在ESP寄存器指向CALL指令的返回位置。当执行RET指令时，程序控制权返回主程序：

```
(gdb) s
area () at functest3.s:47
47          ret
(gdb) print $esp
$4 = (void *) 0xbffff948
(gdb) print $ebp
$5 = (void *) 0x0
(gdb) s
16          addl $4, %esp
(gdb) s
17          movl %eax, result
(gdb) print $esp
$6 = (void *) 0xbffff950
(gdb)
```

控制返回后，ESP值增加4以便删除压入到堆栈中的半径值。现在新的堆栈指针值是0xbffff950，这是函数调用开始前的值。处理过程准备就绪，可以重新开始对下一个半径值的操作。

## 11.4 使用独立的函数文件

使用C样式函数调用的另一个好处是函数是完全自包含的。不需要为访问数据而定义全局内存位置，所以函数中不需要包含.data命令。

这种自由带来了另一个好处：再也不需要在主程序的源代码文件中包含函数源代码了。对于涉及到很多人员的大型项目的程序员来说，这个好处非常有帮助。各个函数可以自包含在它们自己的文件中，并且连接在一起成为最终产品。在编写包含在独立文件中的函数的代码时，程序员会发现继续使用全局变量来传递数据很快就变成了一个问题。每个函数文件都需要跟踪用到的全局变量。

本节演示如何创建独立的函数文件，以及如何汇编它们和把它们与主程序文件连接在一起。

### 11.4.1 创建独立的函数文件

自包含的函数文件和通常创建的主程序文件类似。唯一的区别在于不使用_start段，必须把函数名称声明为全局标签，以便其他程序能够访问它。这是使用.globl命令完成的：

```
.section .text
.type area, @function
.globl area
area:
```

前面的代码片断定义全局标签area，它是前面使用过的area函数的开始。.type命令用于声明area标签指向一个函数的开始。文件area.s中显示完整的函数：

```
# area.s - The area function
.section .text
.type area, @function
.globl area
area:
   pushl %ebp
   movl %esp, %ebp
   subl $4, %esp
   fldpi
```

```
    filds 8(%ebp)
    fmul %st(0), %st(0)
    fmulp %st(0), %st(1)
    fstps -4(%ebp)
    movl -4(%ebp), %eax
    movl %ebp, %esp
    popl %ebp
    ret
```

area.s文件包含area函数的完整源代码，如functest3.s程序中所示。这个版本使用C样式的函数调用，所以从堆栈获得输入值，堆栈也用于定义局部变量。area函数使用EAX寄存器返回输出值。

最后，必须创建主程序，使用一般格式调用外部函数，如functest4.s程序中所示：

```
# functest4.s - An example of using external functions
.section .data
precision:
   .byte 0x7f, 0x00
.section .bss
   .lcomm result, 4
.section .text
.globl _start
_start:
   nop
   finit
   fldcw precision

   pushl $10
   call area
   addl $4, %esp
   movl %eax, result

   pushl $2
   call area
   addl $4, %esp
   movl %eax, result

   pushl $120
   call area
   addl $4; %esp
   movl %eax, result

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

因为主程序中不需要包含area函数的任何代码，所以主程序清单变得更加简短并且不那么混乱。

### 11.4.2 创建可执行文件

创建了函数和主程序之后，必须汇编它们并且把它们连接在一起。必须单独汇编每个程序文件，这样会创建每个文件的目标文件：

```
$ as -gstabs -o area.o area.s
$ as -gstabs -o functest4.o functest4.s
$
```

现在需要把两个目标文件连接在一起创建可执行文件。如果没有连接函数目标文件，连接器就会产生错误：

```
$ ld -o functest4 functest4.o
functest4.o:functest4.s:15: undefined reference to 'area'
functest4.o:functest4.s:20: undefined reference to 'area'
functest4.o:functest4.s:25: undefined reference to 'area'
$
```

连接器无法解析对area函数的调用。为了创建可执行程序，必须在连接器的命令行中包含两个目标文件：

```
$ ld -o functest4 functest4.o area.o
$
```

现在，可以像以前那样在调试器中运行程序，产生相同的结果。

### 11.4.3 调试独立的函数文件

读者也许会注意到在对函数文件和主程序文件进行汇编时，我使用了-gstabs命令行选项。这样确保能够在调试器中查看函数和主程序。有时候，当处理包含许多很长的函数的大型程序时，并不希望这样。

在到达想要调试的函数之前，不必若干次地单步执行长的函数，而是可以选择不调试某个独立函数文件。不必使用-gstabs选项汇编全部函数，而只把它用于想要调试的函数以及主程序。

例如，可以不带-gstabs选项汇编area.s函数，然后在单步执行functest4程序时监视执行情况：

```
$ as -o area.o area.s
$ as -gstabs -o functest3.o functest4.s
$ ld -o functest4 functest4.o area.o
$ gdb -q functest4
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file functest4.s, line 11.
(gdb) run
Starting program: /home/rich/palp/chap11/functest4

Breakpoint 1, _start () at functest4.s:11
11          finit
Current language:  auto; currently asm
(gdb) s
12          fldcw precision
(gdb) s
14          pushl $10
(gdb) s
15          call area
(gdb) s
0x080480b8      31          int $0x80
(gdb) s
16          addl $4, %esp
(gdb)
```

在单步执行主程序的过程中，当执行CALL指令时，它被看作单一指令，调试器中执行的下一条指令是主程序中的下一条指令。area函数中的所有指令被连续处理，并不停止。为了看到这

样能够正确地工作，可以检查返回值以便检验函数的工作：

```
(gdb) print/f $eax
$1 = 314.159271
(gdb)
```

的确，EAX寄存器包含第一次调用area函数得到的结果。

## 11.5 使用命令行参数

与把参数传递给函数相关的主题是把参数传递给程序。程序启动时，一些应用程序需要在命令行中指定输入参数。本节介绍如何在汇编语言程序中利用这一特性。

### 11.5.1 程序剖析

不同的操作系统使用不同的方法把命令行参数传递给程序。在试图解释Linux中如何把命令行参数传递给程序之前，最好首先解释Linux如何从命令行执行程序。

从Linux的Shell提示符运行程序时，Linux系统为要执行的程序在内存中创建一个区域。分配给程序的内存区域可以位于系统物理内存的任何位置。为了使这一过程简化，每个程序都被分配相同的虚拟内存地址。虚拟内存地址由操作系统映射到物理内存地址。

在Linux中，分配给程序运行的虚拟内存地址从地址0x80480000开始，到地址0xbfffffff结束。Linux操作系统按照专门的格式把程序存放在虚拟内存地址中，如图11-7所示。

内存区域中的第一块区域包含汇编程序的所有指令和数据（来自.bss和.data段）。指令不仅包含汇编程序的指令代码，而且包含Linux运行程序的连接过程所需的指令信息。

内存区域中的第二块区域是程序堆栈。就像第2章中讲过的，堆栈从内存区域的底部向下增长。鉴于此，读者会认为程序每次启动时，堆栈指针会被设置为0xbfffffff，但是情况并非如此。在加载程序之前，Linux把一些内容存放到堆栈中，命令行参数就在这里。




图 11-7

### 11.5.2 分析堆栈

程序启动时，Linux把4种类型的信息存放到程序堆栈中：

- 命令行参数（包括程序名称）的数目
- 从shell提示符执行的程序的名称
- 命令行中包含的任何命令行参数
- 在程序启动时的所有当前Linux环境变量

程序名称、命令行参数和环境变量是以空结尾的长度可变的字符串。为了使工作更加简单，Linux不仅把字符串加载到堆栈中，它还把指向每个这些元素的指针加载到堆栈中，所以可以容易地在程序中定位它们。

程序启动时，堆栈的一般布局如图11-8所示。




图 11-8

从堆栈指针（ESP）开始，启动程序所使用的命令行参数的数目作为4字节的无符号整数值被指定。在这之后，指向程序名称位置的4字节指针被存放在堆栈中的下一个位置。再之后，指向每个命令行参数的指针存放在堆栈中（同样，每个指针的长度是4字节）。

可以使用调试器查看这种情况。在调试器中运行functest4程序（前面11.4.3节中提供的），并且在程序启动时监视堆栈。首先，启动程序并且给它提供一个命令行参数（这是在调试器中的运行命令中完成的）：

```
$ gdb -q functest4
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file functest4.s, line 11.
(gdb) run 10
Starting program: /home/rich/palp/chap11/functest4 10

Breakpoint 1, _start () at functest4.s:11
11          finit
Current language:  auto; currently asm
(gdb) print $esp
$1 = (void *) 0xbffff950
(gdb)
```

注意，Starting program这一行表明运行命令中指定的命令行参数。ESP指针显示它指向内存位置0xbffff950。这表示堆栈的顶部。可以使用x命令查看这里的值：

```
(gdb) x/20x 0xbffff950
0xbffff950:  0x00000002   0xbffffa36   0xbffffa57   0x00000000
0xbffff960:  0xbffffa5a   0xbffffa75   0xbffffa95   0xbffffaa7
0xbffff970:  0xbffffabf   0xbffffadf   0xbffffaf2   0xbffffb14
0xbffff980:  0xbffffb26   0xbffffb38   0xbffffb41   0xbffffb4b
0xbffff990:  0xbffffd29   0xbffffd37   0xbffffd58   0xbffffd72
(gdb)
```

x/20x命令显示从指定内存位置开始的前20个字节，格式是十六进制。第一个值表示命令行参数的数目（包括程序名称）。下两个内存位置包含指向稍后存储在堆栈中的程序名称和命令行参数字符串的指针。可以使用x命令和指针地址查看字符串值：

```
(gdb) x/s 0xbffffa36
0xbffffa36:      "/home/rich/palp/chap11/functest4"
(gdb) x/s 0xbffffa57
0xbffffa57:      "10"
(gdb)
```

就像我们期望的，第一个指针指向存储在堆栈区域中的程序名称字符串。第二个指针指向命令行参数的字符串。

小心：记住，所有命令行参数都被指定为字符串，即使它们看上去像是数字，这一点很重要。

在命令行参数之后，4字节的空值被存放到堆栈中，用于把参数和指向环境变量的指针的开始位置分隔开来。同样，可以使用x命令查看存储在堆栈中的一些环境变量：

```
(gdb) x/s 0xbffffa5a
0xbffffa5a:      "PWD=/home/rich/palp/chap11"
(gdb) x/s 0xbffffa75
0xbffffa75:      "http_proxy=http://webproxy:1234"
(gdb) x/s 0xbffffa95
0xbffffa95:      "LC_MESSAGES=en_US"
(gdb) x/s 0xbffffaa7
0xbffffaa7:      "HOSTNAME=test1.blum.lan"
(gdb) x/s 0xbffffabf
0xbffffabf:      "NLSPATH=/usr/share/locale/%l/%N"
(gdb) x/s 0xbffffadf
0xbffffadf:      "LESSKEY=/etc/.less"
(gdb)
```

取决于在Linux环境中加载的是什么应用程序，这里可能加载很多环境变量。

### 11.5.3 查看命令行参数

既然读者已经了解了命令行参数位于堆栈中的什么位置，现在很容易编写简单的程序来访问它们并且列出它们的清单。paramtest1.s程序完成这一工作。

```
# paramtest1.s - Listing command line parameters
.section .data
output1:
   .asciz "There are %d parameters:\n"
output2:
   .asciz "%s\n"
.section .text
.globl _start
_start:
   movl (%esp), %ecx
   pushl %ecx
   pushl $output1
   call printf
   addl $4, %esp
   popl %ecx
   movl %esp, %ebp
   addl $4, %ebp
loop1:
   pushl %ecx
   pushl (%ebp)
   pushl $output2
   call printf
   addl $8, %esp
   popl %ecx
   addl $4, %ebp
   loop loop1
   pushl $0
   call exit
```

paramtest1.s程序首先从堆栈顶部读取命令行参数值的数目，然后把这个值存放在ECX寄存器中（以便使用它作为循环值）。之后，把这个值压入堆栈中，还有C库函数printf要显示的输出文本字符串。

调用printf函数之后，必须把压入堆栈的值清除出去。但是，在这个例子中，这样对我们是有帮助的，因为我们需要在printf函数破坏ECX的值之后恢复它。下面，ESP寄存器中的堆栈指针被复制给EBP指针，以便我们可以遍历这些值，而且不会破坏堆栈指针。EBP指针递增4，跳过命令行参数的计数值，准备好读取第一个参数。

在循环中，ECX的第一个当前值被压入堆栈，以便可以在printf函数之后恢复它。下面，指向第一个命令行参数字符串的指针（位于EBP寄存器指向的内存地址）被压入堆栈，还有printf函数显示它所需要的输出字符串。printf函数返回之后，通过使ESP指针加上8，删除压入堆栈的两个值，并且从堆栈中弹出ECX值。然后，EBP寄存器的值递增4，指向堆栈中下一个命令行参数指针。

对程序进行汇编，并且把它连接到Linux系统中的C库之后，可以使用任意数目的命令行参数测试它：

```
$ ./paramtest1 testing 1 2 3 testing
There are 6 parameters:
./paramtest1
testing
1
2
3
testing
$
```

记住程序名称被认为是第一个参数，第一个命令行参数是第二个参数，等等。初学者的一个常见错误是检查命令行参数的数目是否为零值。它永远都不会为零，因为程序名称总是必须出现在命令行中。

### 11.5.4　查看环境变量

和查看命令行参数一样，可以遍历存储在程序堆栈中的指针并且查看系统的所有环境变量。paramtest2.s程序演示这种做法：

```
# paramtest2.s - Listing system environment variables
.section .data
output:
   .asciz "%s\n"
.section .text
.globl _start
_start:
   movl %esp, %ebp
   addl $12, %ebp
loop1:
   cmpl $0, (%ebp)
   je endit
   pushl (%ebp)
   pushl $output
```

```
    call printf
    addl $12, %esp
    addl $4, %ebp
    loop loop1
endit:
    pushl $0
    call exit
```

不指定命令行参数时，环境变量段从ESP寄存器偏移12的位置开始。环境变量段的结束位置被定义为NULL字符串。把堆栈中的值和零进行比较以便检查这个条件。如果它非零，就使用C函数printf显示指针位置。

对程序进行汇编，并且把它连接到C函数库之后，可以在系统上运行paramtest2程序以便查看实际的环境变量：

```
$ ./paramtest2
TERM=xterm
SHELL=/bin/bash
SSH_CLIENT=192.168.0.4 1698 22
QTDIR=/usr/share/qt3
OLDPWD=/home/rich
SSH_TTY=/dev/pts/0
USER=rich
KDEDIR=/usr
MAIL=/var/mail/rich
PATH=/bin:/sbin:/usr/bin:/usr/sbin:/usr/X11R6/bin:/usr/local/bin:/usr/local/sbin:/u
sr/games
PWD=/home/rich/palp/chap11
LANG=en_US
SHLVL=1
HOME=/home/rich
LOGNAME=rich
SSH_CONNECTION=192.168.0.4 1698 192.168.0.100 22
_=./paramtest2
$
```

系统上存在的环境变量会因运行的是什么应用程序和指定了什么本地设置而不同。读者通过创建新的环境变量并且再次运行程序，可以自己检查环境变量的情况：

```
$ TESTING=/home/rich ; export TESTING
$ ./paramtest2
.
.
.
TESTING=/home/rich
_=./paramtest2
$
```

就像我们期望的，新的环境变量出现在了程序堆栈中。

### 11.5.5 使用命令行参数的范例

在11.5.2节中讲过，在堆栈中命令行参数被存储为字符串值。如果打算将它们作为数字使用，那么要完成转换的工作。

把字符串转换为整数或者浮点值有很多种方式。可以在因特网上搜索一些方便的用于实现转换的汇编语言实用程序。如果不介意在汇编语言程序中使用C库函数，可以使用的标准C转换实用程序有下面这些：

- atoi()：把ASCII字符串转换为短整数值
- atol()：把ASCII字符串转换为长整数值
- atof()：把ASCII字符串转换为双精度浮点值

在调用这些函数之前，指向参数字符串位置的指针必须存放在堆栈中。函数atoi()的结果返回到EAX寄存器中。函数atol()的结果存放在EDX:EAX寄存器对中（因为它需要64位）。函数atof()把它的结果返回到FPU的ST(0)寄存器中。

paramtest3.s程序演示如何读取一个命令行参数，把它转换为整数值，然后从它计算我们过去计算过的面积值：

```
# paramtest3.s - An example of using command line parameters
.section .data
output:
   .asciz "The area is: %f\n"
.section .bss
   .lcomm result, 4
.section .text
.globl _start
_start:
   nop
   finit

   pushl 8(%esp)
   call atoi
   addl $4, %esp
   movl %eax, result
   fldpi
   filds result
   fmul %st(0), %st(0)
   fmul %st(1), %st(0)
   fstpl (%esp)
   pushl $output
   call printf
   addl $12, %esp

   pushl $0
   call exit
```

paramtest3.s程序把指向第一个命令行参数的指针压入堆栈，然后调用C函数atoi()把参数字符串转换为整数值。这个整数值被返回到EAX寄存器中，它被复制到一个内存位置。移动堆栈指针，把参数从堆栈中删除。

下面，使用FLDPI指令把pi值加载到FPU堆栈中，然后使用FILDS指令把存储为4字节整数值的参数传送到FPU堆栈中。之后，进行一般的计算，得到圆的面积。运算完成后，结果位于FPU堆栈寄存器ST(0)中。使用FSTPL指令把这个双精度浮点值压入程序堆栈，以便printf函数进行显示。

对程序进行汇编，并且把它连接到C库之后，可以很容易地使用不同的半径值测试程序：

```
$ ./paramtest3 10
The area is: 314.159265
$ ./paramtest3 2
The area is: 12.566371
$ ./paramtest3 120
The area is: 45238.934212
$
```

现在它开始像个专业的程序了!

## 11.6 小结

本章讨论汇编语言函数这个主题。通过避免多次编写相同的实用程序，函数可以节省开发时间，通过使函数分散到不同程序员可以同时处理的独立文件中，可以使程序设计的过程流水线化。

创建函数时，必须首先确定函数的需求。这包括如何把数据传递给函数，使用什么寄存器和内存位置处理数据，以及如何把结果传递回调用程序。用于在函数和调用程序之间传递数据的基本技术有3种：寄存器、调用程序中定义的通用内存位置和程序堆栈。如果函数用来处理数据的寄存器和内存位置与调用程序使用的相同，就必须要谨慎。在调用函数之前，调用程序必须保存那些其中的值很重要的寄存器，并且在函数返回时恢复它们。

因为在函数和调用程序之间传递数据的可选方式很多，所以设计了一种标准方法帮助简化工作。C样式的函数使用程序堆栈把输入值传递给函数，而且不需要寄存器或者内存位置。在调用函数之前，调用程序把每个输入值压入堆栈。函数不把数据弹出堆栈，而是使用EBP寄存器作为指针，通过间接寻址的方式访问输入值。另外，函数可以在堆栈中保留额外的空间，以便存储函数处理过程中使用的数据元素。这使函数完全自包含成为可能，并且无需使用调用程序提供的任何资源。

因为C样式的函数是完全自包含的，所以可以在单独的源代码文件中创建它们，并且独立于主程序汇编它们。这一特性使多个程序员可以彼此独立地开发不同的函数。必须了解的只是哪些函数完成什么任务，以及应该按照什么顺序把输入值传递给函数。

传递输入参数不仅是函数的需求。很多汇编语言程序也需要处理输入的数据。提供输入数据的一种方法是在执行程序时通过命令行进行。Linux操作系统提供在堆栈区域中把命令行参数传递给程序的标准方式。可以在汇编语言程序中利用这些信息，使用间接寻址的方式访问命令行参数。

在Linux中，当程序启动时，堆栈顶部的第一个值是命令行参数（包括程序名称本身）的数目。堆栈中的下一个值是指向接下来存储在堆栈中的程序名称字符串的指针。之后，存储指向每个命令行参数的指针。每个参数作为空结尾字符串被存储在堆栈中，通过堆栈前面提供的指针访问参数。最后，程序启动时活动的每个系统环境变量都被存放到堆栈区域中。这一特性使程序可以利用环境变量来设置程序的值和选项。

下一章讨论Linux操作系统提供的另一种函数。操作系统内核提供了很多标准函数，比如显示数据、读取数据文件和退出程序。汇编语言程序可以通过使用软件中断来利用这些丰富的功能。和函数调用类似，软件中断把程序的控制权传递给内核函数。通过寄存器传递输入值和输出值。这是从汇编语言程序访问通用Linux函数的便捷方式。

# 第12章 使用Linux系统调用

前一章介绍了如何把汇编语言例程存储为任何程序都可以访问的函数。这一章讲解涉及使用软件中断的另一种访问函数的技术。

大多数操作系统都提供应用程序可以访问的核心函数。Linux也不例外。Linux操作系统的核心提供应用程序可以访问的很多函数，以便容易地访问文件、确定用户和组的权限、访问网络资源以及获得和显示数据。这些函数被称为系统调用（system call）。

本章首先介绍Linux操作系统如何提供系统调用，然后介绍可以使用的资源。之后，提供把系统调用并入汇编语言程序的范例。如果在应用程序中使用很多系统调用，就可能需要进行调试的方法。strace应用程序是非常好的工具，它用于监视活动中的系统调用。本章讲解这个工具，还有通过不同方式使用它的范例。最后，将介绍使用Linux系统调用和C库函数之间的区别。

## 12.1 Linux内核

Linux操作系统的核心是内核。在讨论系统调用之前，了解提供系统调用的操作系统的幕后进行了什么操作是有帮助的。本节简要地描述Linux内核并且解释它如何提供系统调用。如果读者已经熟悉了Linux内核，可以跳过本节。

### 12.1.1 内核组成

内核软件是操作系统的核心。它控制系统的硬件和软件，在必要时分配硬件，并且在需要时执行软件。内核主要有4个责任：

- 内存管理
- 设备管理
- 文件系统管理
- 进程管理

内核提供的系统调用提供了接口，以便程序可以和操作系统的这些组成部分进行交互。了解所有这些组成部分肯定能够帮助理解系统调用试图完成的是什么工作。下面几节介绍这些特性，以及如何使用系统调用提供这些特性的接口。

1. 内存管理

操作系统内核的一个主要功能是内存管理。内核不仅管理服务器上可用的物理内存，它还负责创建和管理“虚拟内存”，或者说在物理上不存在于主板上的内存。

内核通过使用硬盘上的空间完成这个工作，这种空间称为交换空间（swap space），它从硬盘到实际的物理内存来回地交换内存位置。这使系统能够假设可用的内存比物理上存在的内存要多。内存位置被分组为称为页面（page）的块。每个内存页面要么位于物理内存中，要么位于交换空间中。内核必须维护一个表明哪些页面在哪些位置的内存页面表。

内核自动地把一段时间内没有被访问的内存页面复制到硬盘上的交换空间区域。当程序要访问已经被“交换出”的内存页面时，内核必须交换出其他内存页面并且从交换空间交换入所需的页面。

在Linux系统上，通过查看专门的/proc/meminfo文件可以确定虚拟内存的当前状态。在不同的Linux系统之间，meminfo文件的输出是不同的，但是总的来说应该如下：

```
$ cat /proc/meminfo
        total:    used:    free:  shared: buffers:  cached:
Mem:  129957888 119554048 10403840        0 34099200 37548032
Swap: 254943232  2945024 251998208
MemTotal:       126912 kB
MemFree:         10160 kB
MemShared:           0 kB
Buffers:         33300 kB
Cached:          36668 kB
Active:          23116 kB
Inact_dirty:     46064 kB
Inact_clean:       788 kB
Inact_target:       12 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       126912 kB
LowFree:         10160 kB
SwapTotal:      248968 kB
SwapFree:       246092 kB
$
```

第一行显示用于查看/proc/meminfo文件的Linux命令。第三行显示这台Linux服务器具有128MB物理内存。还显示当前正在使用的内存大约有10MB。下一行显示这一系统上可用的交换空间大约有252MB。

运行在Linux系统上的每个进程都有其自己的私有内存区域。一个进程不能访问另一个进程正在使用的内存。没有进程能够访问内核进程使用的内存。为了便于进行数据共享，可以创建共享内存段。多个进程可以读取和写入通用共享内存区域。内核必须维护和管理共享内存区域。可以使用ipcs命令查看系统上当前的共享内存段：

```
# ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 0          root       600        1056768    5          dest
0x00000000 32769      apache     600        46084      5          dest
0x00000000 65538      root       600        1056768    5          dest
0x00000000 10289155   root       600        520192     5          dest
0x00000000 131076     apache     600        46084      5          dest
0x0052e2ca 163845     postgres   700        144        1
0x0052e2c1 196614     postgres   600        1104896    1
0x0052e2c7 229383     postgres   600        66060      1

#
```

上面表明使用-m选项的ipcs命令只显示共享内存段。每个共享内存段都有创建这个段的所

有者。每个段还有标准的UNIX权限设置，它设置这个段对其他用户的可用性。key值用于使其他用户可以访问共享内存段。

2. 设备管理

内核的另一个责任是硬件管理。必须与Linux系统进行通信的任何设备都需要插入到内核代码中的驱动代码。驱动代码使内核可以在通用接口到设备之间来回传递数据。有两种方法用于把设备驱动代码插入到Linux内核中：

- 把驱动代码编译到内核代码中
- 把驱动代码插入到正在运行的内核中

以前，插入设备驱动代码的唯一途径是重新编译内核。每次把新的设备添加到系统中时，都需要重新编译内核代码。随着Linux内核支持的硬件越来越多，这种方式的效率就越低。

把驱动代码插入到正在运行的内核中的更好的方法被开发出来了。这就是内核模块（kernel module）的概念，它允许把驱动代码插入到正在运行的内核中，当设备使用完毕时也可以从内核删除驱动代码。

在UNIX服务器上，硬件设备被标识为特殊的设备文件。有3种不同类别的设备文件：

- 字符
- 块
- 网络

字符文件代表一次只处理一个字符数据的设备。大多数类型的终端接口被创建为字符文件。块文件代表一次处理一大块数据的设备，比如磁盘驱动器。网络文件类型代表使用包发送和接收数据的设备。这包括网卡和特殊的回送设备，回送设备允许Linux系统使用通用网络编程协议和自身进行通信。

在文件系统中，设备文件被创建为节点。每个节点都具有对Linux内核标识它的唯一的数字对。这个数字对包含主要设备号和次要设备号。相似的设备被分配到相同的主设备号组中。次设备号用于在主设备号相同的设备之间标识设备。下面是Linux系统上一些设备的清单：

```
# ls -al hda*
brw-rw----    1 root     disk       3,   0 Apr 14  2001 hda
brw-rw----    1 root     disk       3,   1 Apr 14  2001 hda1
brw-rw----    1 root     disk       3,  10 Apr 14  2001 hda10
brw-rw----    1 root     disk       3,   2 Apr 14  2001 hda2
brw-rw----    1 root     disk       3,   3 Apr 14  2001 hda3
brw-rw----    1 root     disk       3,   4 Apr 14  2001 hda4
brw-rw----    1 root     disk       3,   5 Apr 14  2001 hda5
# ls -al ttyS*
[root@test2 /dev]# ls -al ttyS* | more
crw-rw----    1 root     uucp       4,  64 Apr 14  2001 ttyS0
crw-rw----    1 root     uucp       4,  65 Apr 14  2001 ttyS1
crw-rw----    1 root     uucp       4,  74 Apr 14  2001 ttyS10
crw-rw----    1 root     uucp       4, 164 Apr 14  2001 ttyS100
crw-rw----    1 root     uucp       4, 165 Apr 14  2001 ttyS101
crw-rw----    1 root     uucp       4, 166 Apr 14  2001 ttyS102
crw-rw----    1 root     uucp       4, 167 Apr 14  2001 ttyS103
#
```

上面使用ls命令显示hda和ttyS设备的一些条目。hda设备是系统上第一个IDE硬盘驱动器的

分区，ttyS设备是标准IBM PC的COM端口。这个清单显示这个Linux系统上创建了若干hda设备。实际上没有使用所有这些设备，但是它们被创建了以防管理员需要它们。类似地，这个清单显示若干ttyS设备也被创建了。

清单中的第5列是主设备节点号。注意所有hda设备都具有相同的主设备节点3，并且所有ttyS设备都使用4。第6列是次设备节点号。一个主设备节点号中的每个设备都具有它自己唯一的次设备节点号。这是内核分别访问每个设备的方式。

第一列表示设备文件的权限。权限的第一个字符表示文件的类型。注意IDE硬盘驱动器文件都被标记为块（b）文件，COM端口设备文件被标记为字符（c）文件。

3. 文件系统管理

和一些其他的操作系统不同，Linux内核可以支持不同类型的文件系统，对硬盘驱动器读取和写入数据。现在，Linux系统上可用的有15种不同类型的文件系统。内核必须被编译为支持系统将使用的所有文件系统类型。下表介绍Linux系统上可用的标准文件系统。

| 文件系统 | 描　述 |
|---|---|
| affs | Amiga文件系统 |
| ext | Linux扩展文件系统 |
| ext2 | 第二扩展文件系统 |
| ext3 | 第三扩展文件系统 |
| hpfs | OS/2高性能文件系统 |
| iso9660 | ISO 9660文件系统（CD-ROM） |
| minix | MINIX文件系统 |
| msdos | Microsoft的FAT16 |
| ncp | Netware文件系统 |
| proc | 访问系统信息 |
| reiserfs | 日志文件系统 |
| sysv | 旧式的UNIX文件系统 |
| ufs | BSD文件系统 |
| umsdos | 驻留于MS-DOS上的类UNIX文件系统 |
| vfat | Windows 95文件系统（fat32） |

Linux系统访问的任何硬盘驱动器都必须使用上表中列出的文件系统类型之一进行格式化。格式化Linux文件系统和格式化Microsoft Windows类型的磁盘类似。在能够使用磁盘存储信息之前，操作系统必须在磁盘上构建必须的文件系统信息。

Linux内核使用虚拟文件系统（Virtual File System，VFS）与每种文件系统进行交互。这为内核与任何类型的文件系统的通信提供标准的接口。挂装和使用每种文件系统时，VFS把信息缓存在内存中。

内核提供系统调用，帮助使用VFS来管理和访问每种不同文件系统上的文件。单一系统调用可以用于访问任何文件系统类型上的文件。

4. 进程管理

Linux操作系统把程序作为进程进行管理。内核控制如何在系统中管理进程。内核创建的第一个进程（称为init进程）启动系统上的所有其他进程。内核启动时，init进程被加载到虚拟内存

中。每个进程启动时，为它分配虚拟内存中的区域，用于存储数据和系统将执行的代码。

一些Linux实现包含在引导时自动启动的终端进程的列表。每个终端进程都提供一个访问点，用于交互地登录到Linux系统。init进程启动时，它读取文件/etc/inittabs以便确定它必须在系统上启动什么终端进程。

Linux操作系统使用一种利用运行级别（run level）的init系统。运行级别用于指示init进程只运行特定类型的进程。在Linux操作系统上有5个init运行级别。

在运行级别1，只启动基本的系统进程，还有一个控制台终端进程。这称为单一用户模式（single-user mode）。单一用户模式经常用于文件系统维护。标准的init运行级别是3。在这个运行级别，启动大多数应用程序软件（比如网络支持软件）。在Linux中，另一个常用的运行级别是5。在这个运行级别上启动X Window软件。注意Linux系统如何通过控制init运行级别来控制全面的系统功能。通过把运行级别从3改动为5，系统可以从基于控制台的系统改变为高级的、图形化的X Window系统。

为了查看Linux系统上当前活动的进程，可以使用ps命令。ps命令的格式是：

```
ps options
```

其中options是可以修改ps命令的输出的选项列表。下表介绍可用的选项。

| 选　项 | 描　述 |
|---|---|
| l | 使用长格式进行显示 |
| u | 使用用户格式（显示用户名称和启动时间） |
| j | 使用作业格式（显示进程gid和sid） |
| s | 使用信号格式 |
| v | 使用vm格式 |
| m | 显示内存信息 |
| f | 使用“森林型”格式（将进程显示为树型） |
| a | 显示其他用户的进程 |
| x | 显示不带控制终端的进程 |
| S | 显示子CPU和时间以及页面错误 |
| c | 用于task_struct的命令名称 |
| e | 在命令行和la＋后显示环境 |
| w | 使用宽输出格式 |
| h | 不显示标题 |
| r | 只显示正在运行的进程 |
| n | 显示USER和WCHAN的数字输出 |
| txx | 显示终端ttyxx控制的进程 |
| O | 使用排序键k1、k2等等对进程清单进行排序 |
| pids | 只显示指定的pid |

很多选项可以用于修改ps命令的输出。下面是ps清单输出的一个例子：

```
# ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        S      1:16 init [5]
    2 ?        SW     0:00 [keventd]
    3 ?        SW   131884:22 [kapm-idled]
```

```
    4 ?        SW     5:54 [kswapd]
    5 ?        SW     0:00 [kreclaimd]
    6 ?        SW     2:32 [bdflush]
    7 ?        SW     2:39 [kupdated]
    8 ?        SW<    0:00 [mdrecoveryd]
   63 ?        S      0:00 open -w -s -c 12 /sbin/Monitor-NewStyle-Categorizing-WsLib
   68 tty12    S      0:03 /sbin/Monitor-NewStyle-Categorizing-WsLib
  565 ?        SW     0:00 [khubd]
  663 ?        S      0:00 /sbin/cardmgr
  814 ?        SW     0:00 [eth0]
  876 ?        S      0:11 syslogd -m 0
  885 ?        S      0:00 klogd -2
  981 ?        S      0:00 xinetd -reuse -pidfile /var/run/xinetd.pid
 1292 ?        S      0:23 httpd-perl -f /etc/httpd/conf/httpd-perl.conf -DPERLPROXIED
 1299 ?        S      0:00 httpd-perl -f /etc/httpd/conf/httpd-perl.conf -DPERLPROXIED
 1300 ?        S      0:00 httpd-perl -f /etc/httpd/conf/httpd-perl.conf -DPERLPROXIED
 1301 ?        S      0:00 httpd-perl -f /etc/httpd/conf/httpd-perl.conf -DPERLPROXIED
 1302 ?        S      0:00 httpd-perl -f /etc/httpd/conf/httpd-perl.conf -DPERLPROXIED
 1307 ?        S      0:55 httpd -DPERLPROXIED -DHAVE_SSL -DHAVE_PROXY -DHAVE_ACCESS
 1293 ?        S      0:00 pickup -l -t fifo
 2359 ?        S      0:00 /usr/sbin/sshd
 2360 pts/0    S      0:00 -bash
 2422 pts/0    S      0:00 su
 2423 pts/0    S      0:00 bash
 2966 pts/0    R      0:00 ps ax
#
```

第一行显示命令行中输入的ps命令。使用a和x选项使输出显示系统上正在运行的所有进程。输出的第一列显示进程的进程ID（即PID）。第3行显示内核启动的init进程。分配给init进程的PID是1。按照数字顺序为init进程之后启动的所有其他进程分配PID。两个进程不能具有相同的PID。

第3列显示进程的当前状态。下表介绍可能的进程状态代码。

| 代 码 | 描 述 |
|---|---|
| D | 不可中断的睡眠 |
| L | 进程具有在内存中锁定的页面 |
| N | 低优先权的任务 |
| R | 可运行 |
| S | 进程要求页面替换（正在睡眠） |
| T | 被跟踪或者停止 |
| Z | 死亡（僵）的进程 |
| W | 无驻留页面的进程 |
| < | 高优先权的进程 |

最后一列显示进程名称。括在[]括号中的进程是由于不活动已经被从内存交换出到了磁盘交换空间的进程。可以发现一些进程已经被交换出了，但是大多数正在运行的进程没有被交换出。

### 12.1.2 Linux内核版本

系统运行哪个内核版本直接影响应用程序可以使用哪些系统调用。每个新的内核版本产生时，都会创建额外的系统调用，在不同的领域中帮助程序员。

Linux内核开发的进展步伐非常快。Linus Torvalds严格地控制Linux内核，虽然他接受来自任何人、任何地方的改动要求。多年以来，Linux内核设计中的很多高级特性被不断开发出来，比如增加了模块。

内核开发人员使用严格的版本控制系统。内核版本的格式是

```
linux-a.b.c
```

其中a是主版本号，b是次版本号，c是补丁号（通常有中间的补丁，所以c值也可以是c.d，其中d是中间补丁号）。已经建立的惯例是：奇数的次版本号被认为是试验性的版本，而偶数的次版本号被认为是稳定的产品型版本。

到编写本书时，Linux内核的最新的稳定产品型版本是2.6.8.1，最新的试验性的版本是2.4.2-pre1。虽然2.4.1版本是最新的内核版本，但是大多数Linux版本还没有使用这个内核版本。

为了确定所在Linux系统使用的内核版本，可以使用带有-a选项的uname命令。在我的Mandrake Linux 8.0系统上，这个命令生成下面的输出：

```
$ uname -a
Linux test.blum.lan 2.4.3-20mdk #1 Sun Apr 15 23:03:10 CEST 2001 i686 unknown
$
```

清单中的第3个项目就是内核版本。对于这个系统，使用的核心Linux内核是2.4.3版本，还附加了一些Mandrake扩展。

## 12.2 系统调用

既然读者已经了解了Linux内核，现在该研究内核系统调用了。内核提供了很多系统调用，了解如何查找和使用它们对汇编语言程序设计会有所帮助。本节介绍如何在程序中利用Linux系统调用。

### 12.2.1 查找系统调用

前面讲过，程序员可以使用的系统调用有很多。通常随着每个新的内核版本的发布，都会有新的系统调用添加到这个清单中。很容易确定系统上可用的系统调用有哪些——只需查看内核即可。

如果Linux系统已经针对一个程序设计开发环境配置过了（如果正在汇编程序，那么很可能就是如此），系统调用在下面的文件中定义：

```
/usr/include/asm/unistd.h
```

unistd.h文件包含内核中每个可用的系统调用的定义。这个文件的前面几行如下：

```
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#define __NR_exit                1
```

```
#define __NR_fork              2
#define __NR_read              3
#define __NR_write             4
#define __NR_open              5
#define __NR_close             6
#define __NR_waitpid           7
#define __NR_creat             8
#define __NR_link              9
#define __NR_unlink           10
#define __NR_execve           11
#define __NR_chdir            12
#define __NR_time             13
#define __NR_mknod            14
#define __NR_chmod            15
#define __NR_lchown           16
```

每个系统调用都被定义为一个名称（前面加上_NR_），和它的系统调用号。在下一节中将看到，系统调用号对汇编语言程序员是至关重要的，因为它是汇编程序引用系统调用的途径。

### 12.2.2 查找系统调用定义

我的Linux系统上的unistd.h文件中定义了221个系统调用。如果读者有兴趣使用任何系统调用，就必须了解汇编语言程序的调用格式。

在系统上的系统调用清单中找到需要的系统调用之后，通常可以在系统的man页中找到定义（如果已经安装了的话）。man页的第2部分包含所有可用的系统调用的定义。

如果系统没有安装man页，可以参考Linux发布文档来安装它们。

为了访问系统调用定义，可以从命令提示符使用man命令：

```
# man 2 exit
```

命令行中的2指定man页的第2部分。不要忘记包含它，因为一些系统调用也包含在man页第1部分中列出的Shell命令。如果不在命令行中指定2，就会显示它们而不是系统调用定义。exit选项指定要获得信息的系统调用名称。man页包含4个主要部分：

- Name（名称）：显示这个系统调用的名称
- Synopsis（提要）：显示如何使用这个系统调用
- Description（描述）：对这个系统调用的简要描述
- Return Value（返回值）：系统调用完成时返回的值

下面是系统调用exit的man页的内容：

```
NAME
       _exit - terminate the current process

SYNOPSIS
       #include <unistd.h>
       void _exit(int status);

DESCRIPTION
       _exit terminates the calling process immediately. Any open
       file descriptors belonging to the process are closed;  any
```

```
       children  of the process are inherited by process 1, init,
       and the process's parent is sent a SIGCHLD signal.

       status is returned to the parent process as the  process's
       exit  status,  and  can be collected using one of the wait
       family of calls.

RETURN VALUE
       _exit never returns.
```

提要部分是为C程序员编写的，但是汇编语言程序员也可以从这里获得帮助。这个系统调用的提要显示它使用单个整数输入值（括号中的值），并且不生成返回值。

### 12.2.3 常用系统调用

虽然可以选择的系统调用有很多，但是大多数常见的程序设计功能只需要几个系统调用。本节介绍一些比较常用的系统调用，按照内核特性分组。

下表介绍内存访问内核系统调用。

| 系统调用 | 描　述 |
|---|---|
| brk | 改变数据段长度 |
| mlock | 禁止对内存部分进行分页 |
| mlockall | 禁止对调用进程进行分页 |
| mmap | 把文件或者设备映射到内存中 |
| mprotect | 控制对内存区域的许可访问 |
| mremap | 重新映射虚拟内存地址 |
| msync | 同步文件和内存映射 |
| munlock | 允许对内存部分进行分页 |
| munlockall | 允许对调用进程进行分页 |
| munmap | 取消文件或者设备在内存中的映射 |

下表介绍常用的设备访问内核系统调用。

| 系统调用 | 描　述 |
|---|---|
| access | 检查设备的权限 |
| chmod | 改变设备的权限 |
| chown | 改变设备的所有关系 |
| close | 关闭设备文件描述符 |
| dup | 复制设备文件描述符 |
| fcntl | 操作文件描述符 |
| fstat | 获得设备的状态 |
| ioctl | 控制设备的参数 |
| link | 把新的名称分配给文件描述符 |
| lseek | 重新定位读取/写入文件偏移量 |
| mknod | 为设备创建新的文件描述符 |
| open | 为设备或者文件打开/创建文件描述符 |
| read | 读取设备文件描述符 |
| write | 写入设备文件描述符 |

下表介绍文件系统系统调用。

| 系统调用 | 描 述 |
|---|---|
| chdir | 改变工作目录 |
| chroot | 改变根目录 |
| flock | 在打开的文件上应用或者删除建议锁（advisory lock） |
| statfs | 获得文件系统的统计数据 |
| getcwd | 获得当前工作目录 |
| mkdir | 创建目录 |
| rmdir | 删除目录 |
| symlink | 生成文件的新名称 |
| umask | 设置文件创建掩码 |
| mount | 挂装和卸载文件系统 |
| swapon | 开始内存和文件系统的交换 |
| swapoff | 停止内存和文件系统的交换 |

最后，下表介绍进程系统调用。

| 系统调用 | 描 述 |
|---|---|
| acct | 打开或者关闭进程计数 |
| capget | 获得进程功能 |
| capset | 设置进程功能 |
| clone | 创建子进程 |
| execve | 执行程序 |
| exit | 终止当前进程 |
| fork | 创建子进程 |
| getgid | 获得组标识 |
| getpgrp | 获得/设置进程组 |
| getppid | 获得进程标识 |
| getpriority | 获得程序调度优先权 |
| getuid | 获得用户标识 |
| kill | 发送信号杀死进程 |
| nice | 改变进程优先权 |
| vfork | 创建子进程并且阻塞父进程 |

## 12.3 使用系统调用

在汇编语言程序中使用系统调用是复杂的。本节介绍如何使用系统调用，并且讲解如何在范例程序中使用它们。

### 系统调用格式

在本书前面的例子中读者已经看到，为了启动系统调用，需要使用INT指令。Linux系统调用位于中断0x80。执行INT指令时，所有操作转移到内核中的系统调用处理程序。系统调用完成时，执行转移回INT指令之后的下一条指令（当然，除非执行了exit系统调用）。

下面几节介绍如何创建系统调用。

1. 系统调用值

如果读者从头开始阅读本书，从前面的范例程序中已经了解了EAX寄存器用于保存系统调用值。这个值定义使用内核支持的系统调用中的哪个系统调用。

unistd.h文件中系统调用名称旁边列出的整数就是系统调用值（value）。每个系统调用都被分配了唯一的数字以便标识它。在执行INT指令之前，期望的值被传送到EAX寄存器中。本书中已经使用过的exit系统调用的例子很好地展示了这种情况：

```
movl $1, %eax
int 0x80
```

unistd.h文件中exit系统调用的定义如下：

```
#define __NR_exit  1
```

这表明执行exit系统调用的系统调用值是1。使用标准的MOVL指令把这个值加载到EAX寄存器中。把系统调用值存放到EAX寄存器中之后，执行INT指令，使用内核系统调用的向量值（0x80）。

2. 系统调用输入值

在C样式的函数中，输入值被存放在堆栈中；系统调用与之不同，需要输入值被存放在寄存器中。每个输入值要按照特定的顺序存放到寄存器中。把错误的输入值存放在错误的寄存器中可能导致灾难性的结果。

读者已经看到了EAX寄存器用于保存要执行的系统调用值。不能使用EIP、EBP和ESP寄存器，因为这样可能对程序操作产生有害影响。这样就只剩下5个寄存器可以用于保存输入值。

前面讲过，输入值按照什么顺序存放到寄存器中是很重要的。系统调用期望的输入值顺序如下：

- EBX（第1个参数）
- ECX（第2个参数）
- EDX（第3个参数）
- ESI（第4个参数）
- EDI（第5个参数）

需要超过6个输入参数的系统调用使用不同的方法把参数传递给系统调用。EBX寄存器用于保存指向输入参数的内存位置的指针，输入参数按照连续的顺序存储。系统调用使用这个指针访问内存位置以便读取参数。

下一个技巧是决定系统调用函数中的哪个参数属于哪个顺序。为了描述这种情况，我们最好研究一个例子。

系统调用write()用于把数据写入文件描述符。如果查看man页，这个系统调用的提要如下：

```
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);
```

按照参数出现在提要中的次序从左至右设置参数号。第一个参数（fd）是代表输出设备的文件描述符的整数值。第二个参数（buf）是指向要写入设备的字符串的指针。第三个参数（count）是要写入的字符串的长度。

使用这一规范，输入值应该被分配到下面的寄存器中：

- EBX：整数文件描述符
- ECX：指向要写入的字符串的指针（内存地址）
- EDX：要写入的字符串的长度

syscalltest1.s程序显示使用这个系统调用的一个例子：

```
# syscalltest1.s - An example of passing input values to a system call
.section .data
output:
   .ascii "This is a test message.\n"
output_end:
   .equ len, output_end - output
.section .text
.globl _start
_start:
   movl $4, %eax
   movl $1, %ebx
   movl $output, %ecx
   movl $len, %edx
   int $0x80

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

现在读者可能已经熟悉了syscalltest1.s程序中的大部分内容。系统调用write()的系统调用值（4）被存放在EAX寄存器中。然后把系统调用需要的输入值存放在适当的寄存器中。

输出位置的文件描述符值存放在EBX寄存器中。Linux系统包含3种专门的文件描述符：

- 0（STDIN）：终端设备的标准输入（一般是键盘）
- 1（STDOUT）：终端设备的标准输出（一般是终端屏幕）
- 2（STDERR）：终端设备的标准错误输出（一般是终端屏幕）

syscalltest1.s程序使用STDOUT文件描述符把文本显示在终端屏幕上。下一个输入值是要显示的字符串。注意指定内存位置output时使用直接寻址模式。这样把标签output指向的实际内存位置地址存放在ECX寄存器中，而不是存放内存位置中的值。

最后一个输入参数是要显示的字符串的长度。syscalltest1.s程序没有硬编码一个静态长度值，而是使用常用的汇编语言技巧确定字符串的长度。前面讲过，标签output用于声明字符串开始的内存位置。在.ascii命令后面马上使用另一个标签。标签output_end声明了紧跟在字符串末尾后面的位置。虽然这个位置中什么都没有存储，但是现在可以使用这个标签本身指向字符串的末尾。通过将这两个标签相减，.equ命令定义了长度值：

```
.equ len, output_end - output
```

现在可以在整个程序中使用标签len表示字符串的长度。记住标签len被作为立即值对待，所

以使用它时必须在它前面加上美元符号:

```
movl $len, %edx
```

这条指令把字符串的长度加载到EDX寄存器中。现在所有寄存器都被加载了内容，系统调用准备就绪，可以使用INT指令执行它。

汇编和连接程序之后（记住系统调用不需要任何C函数，所以不必与C库连接），可以测试它:

```
$ ./syscalltest1
This is a test message.
$
```

3. 系统调用返回值

如系统调用write的提要所示，很多系统调用在完成之后会返回值。在系统调用write的例子中，它返回被写入到文件描述符的字符串的长度，或者如果调用失败的话，返回负值。

系统调用的返回值存放在EAX寄存器中。程序员负责检查EAX寄存器中的这个值，特别是在失败的情况下。

一定要了解返回值的数据类型。一些系统调用使用怪异的数据类型处理数据。前面的系统调用write就是演示这种情况的恰当例子。

根据提要，系统调用write返回ssize_t数据类型的值。ssize_t数据类型不是一种可用的标准汇编语言数据类型。ssize_t数据类型是带符号整数的typedef（或同义词），如果操作成功，它表示写入到文件描述符的字符的数目，如果发生错误则是-1。

syscalltest2.s程序演示如何处理系统调用的返回值:

```
# syscalltest2.s - An example of getting a return value from a system call
.section .bss
   .lcomm pid, 4
   .lcomm uid, 4
   .lcomm gid, 4
.section .text
.globl _start
_start:
   movl $20, %eax
   int $0x80
   movl %eax, pid

   movl $24, %eax
   int $0x80
   movl %eax, uid

   movl $47, %eax
   int $0x80
   movl %eax, gid
end:
   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

syscalltest2.s程序使用下表中介绍的3个单独的系统调用。

| 系统调用值 | 系统调用 | 描 述 |
|---|---|---|
| 20 | getpid | 获得正在运行的程序的进程ID |
| 24 | getuid | 获得运行此程序的人的用户ID |
| 47 | getgid | 获得运行此程序的人的组ID |

把每个系统调用值传送到EAX寄存器并且执行INT指令之后，EAX寄存器中的返回值被存放到适当的内存位置中。

汇编和连接程序之后，可以在调试器中运行它以便查看这些值。创建end标签以便提供方便的断点位置，从而可以在程序退出之前查看值。下面是在我的系统上的输出情况：

```
$ gdb -q syscalltest2
(gdb) break *end
Breakpoint 1 at 0x8048098: file syscalltest2.s, line 21.
(gdb) run
Starting program: /home/rich/palp/chap12/syscalltest2

Breakpoint 1, end () at syscalltest2.s:21
21          movl $1, %eax
Current language:  auto; currently asm
(gdb) x/d &pid
0x80490a4 <pid>:        4758
(gdb) x/d &uid
0x80490a8 <uid>:        501
(gdb) x/d &gid
0x80490ac <gid>:        501
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit
$
```

可以使用x/d调试器命令，把内存位置pid、uid和gid中的值显示为整数值。虽然进程ID对于正在运行的程序是唯一的，但是可以使用Shell命令id检查uid和gid的值：

```
$ id
uid=501(rich) gid=501(rich) groups=501(rich), 22(cdrom), 43(usb), 80(cdwriter),
81(audio), 503(xgrp)
$
```

通过syscalltest2程序，可以确定使用的uid是501，使用的gid也是501。

## 12.4 复杂的系统调用返回值

有时候系统调用返回涉及C样式结构的复杂数据。在汇编语言程序中使用它们时，有时候难以决定如何处理返回的C结构，以及如何把它转换为汇编语言程序能够处理的数据类型。

本节演示如何使用返回一个数据结构的系统调用，并且介绍可以用来处理返回的数据的一种方法。

### 12.4.1 sysinfo系统调用

系统调用sysinfo可以用于返回关于系统如何配置以及有什么可用资源的信息。系统调用sysinfo的man页如下:

```
NAME
       sysinfo - returns information on overall system statistics

SYNOPSIS
       #include <sys/sysinfo.h>

       int sysinfo(struct sysinfo *info);
```

系统调用sysinfo使用单一输入值，它指向保存包含返回数据的结构的内存位置。man页还显示了这种结构的情况:

```
struct sysinfo {
           long uptime;             /* Seconds since boot */
           unsigned long loads[3];  /* 1, 5, and 15 minute load averages */
           unsigned long totalram;  /* Total usable main memory size */
           unsigned long freeram;   /* Available memory size */
           unsigned long sharedram; /* Amount of shared memory */
           unsigned long bufferram; /* Memory used by buffers */
           unsigned long totalswap; /* Total swap space size */
           unsigned long freeswap;  /* swap space still available */
           unsigned short procs;    /* Number of current processes */
           unsigned long totalhigh; /* Total high memory size */
           unsigned int mem_unit;   /* Memory unit size in bytes */
           char _f[20-2*sizeof(long)-sizeof(int)]; /* Padding for libc5 */
}
```

每个系统值被返回到结构内的特定位置中。必须在一个内存位置创建这个结构，以便值可以被返回到这里:

```
.section .data
result:
uptime:
   .int 0
load1:
   .int 0
load5:
   .int 0
load15:
   .int 0
totalram:
   .int 0
freeram:
   .int 0
sharedram:
   .int 0
bufferram:
   .int 0
totalswap:
   .int 0
freeswap:
   .int 0
```

```
totalhigh:
   .int 0
memunit:
   .int 0
```

读者也许会注意到在数据定义的开头有两个标签。当对程序进行汇编时，它们都指向相同的内存位置。标签result可以用于引用整个结构，标签uptime可以用于引用结构中的第一个值。

### 12.4.2 使用返回结构

定义好返回结构之后，就可以在汇编语言程序中使用这些信息。下面显示的sysinfo.s程序使用这个结构从系统调用sysinfo获得系统信息：

```
# sysinfo.s - Retrieving system information via kernel system calls
.section .data
result:
uptime:
   .int 0
load1:
   .int 0
load5:
   .int 0
load15:
   .int 0
totalram:
   .int 0
freeram:
   .int 0
sharedram:
   .int 0
bufferram:
   .int 0
totalswap:
   .int 0
freeswap:
   .int 0
procs:
   .byte 0x00, 0x00
totalhigh:
   .int 0
memunit:
   .int 0
.section .text
.globl _start
_start:
   nop
   movl $result, %ebx
   movl $116, %eax
   int $0x80

   movl $0, %ebx
   movl $1, %eax
   int $0x80
```

系统调用sysinfo的系统调用值被存放在EAX寄存器中，标签result的内存位置存储在EBX寄

存器中作为输入值。执行INT指令之后，返回结构值被加载到这个内存位置中，各个内存标签可以用于引用每个单独的值。

### 12.4.3 查看结果

系统调用完成之后，可以通过数据元素的标签获得各个数据元素。使用调试器并且在执行系统调用之后查看值可以演示这种情况：

```
(gdb) x/d &uptime
0x8049090 <uptime>:     27421854
(gdb) x/d &load1
0x8049094 <load1>:      3296
(gdb) x/d &load5
0x8049098 <load5>:      3200
(gdb) x/d &load15
0x804909c <load15>:     448
(gdb) x/d &totalram
0x80490a0 <totalram>:   129957888
(gdb) x/d &freeram
0x80490a4 <freeram>:    11579392
(gdb) x/d &sharedram
0x80490a8 <sharedram>:  0
(gdb) x/d &bufferram
0x80490ac <bufferram>:  31346688
(gdb) x/d &totalswap
0x80490b0 <totalswap>:  254943232
(gdb) x/d &freeswap
0x80490b4 <freeswap>:   251998208
(gdb) x/d &procs
0x80490b8 <procs>:      53
(gdb) x/d &totalhigh
0x80490ba <totalhigh>:  0
(gdb) x/d &memunit
0x80490be <memunit>:    0
```

## 12.5 跟踪系统调用

在汇编语言程序中使用系统调用时，能够查看程序运行时发生的情况经常是有帮助的。Linux系统提供了strace程序，它在应用程序运行时跟踪其中使用的系统调用。使用strace程序可用的选项有很多。本节介绍strace程序并且演示如何使用它查找汇编语言系统调用的问题。

### 12.5.1 strace程序

strace程序截取程序发出的系统调用并且显示它们以供查看。被跟踪的程序可以是从strace命令运行的，也可以是系统上已经运行的进程。如果具有适当的权限，就可以研究现有的进程并且监视发出的系统调用。在调试汇编语言和高级语言的程序时它是价值无法估量的工具。

通常，只需使用strace程序的默认选项生成关于程序的必要信息即可：

```
$ strace ./syscalltest2
execve("./syscalltest2", ["./syscalltest2"], [/* 38 vars */]) = 0
```

```
getpid()                                = 7616
getuid()                                = 501
getgid()                                = 501
_exit(0)                                = ?
$
```

上面的输出显示了syscalltest2程序发出的所有系统调用，按照应用程序执行系统调用的顺序排列。左侧显示系统调用名称，右侧显示系统调用生成的返回值。系统调用execve显示操作系统shell如何运行这个程序。这包括堆栈中的命令行参数和环境变量（参见第11章）。

命令行参数-c在程序执行之后创建一个报告，概述发出的所有系统调用，以及每个系统调用花费了多长时间：

```
$ strace -c ./syscalltest2
execve("./syscalltest2", ["./syscalltest2"], [/* 38 vars */]) = 0
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 60.00    0.000006           6         1           getpid
 20.00    0.000002           2         1           getuid
 20.00    0.000002           2         1           getgid
------ ----------- ----------- --------- --------- ----------------
100.00    0.000010                     3           total
$
```

在这个简单的例子中，没有太多的数据值得仔细研究。

### 12.5.2 高级strace参数

strace程序的基本操作生成很多数据，可以使用命令行参数调整strace程序以便适合特定的应用程序。下表介绍可用的命令行参数。

| 参 数 | 描 述 |
|---|---|
| -c | 统计每个系统调用的时间、调用和错误 |
| -d | 显示strace的一些调试输出 |
| -e | 指定输出的过滤表达式 |
| -f | 在创建子进程的时候跟踪它们 |
| -ff | 如果写入到输出文件，则把每个子进程写入到单独的文件中 |
| -i | 显示执行系统调用时的指令指针 |
| -o | 把输出写入到指定的文件 |
| -p | 附加到由PID指定的现有进程 |
| -q | 抑制关于附加和分离的消息 |
| -r | 对每个系统调用显示一个相对的时间戳 |
| -t | 把时间添加到每一行 |
| -tt | 把时间添加到每一行，包括微秒 |
| -ttt | 添加epoch形式的时间（从1970年1月1日开始的秒数），包括微秒 |
| -T | 显示每个系统调用花费的时间 |
| -v | 显示系统调用信息的不经省略的版本（详细的） |
| -x | 以十六进制格式显示所有非ASCII字符 |
| -xx | 以十六进制格式显示所有字符串 |

大多数命令行参数的含义都非常清楚。-e参数很方便，因为它可以用于只显示系统调用的子集，而不是查看全部。-e参数的格式如下：

```
trace=call_list
```

其中call_list是以逗号分隔的希望跟踪的系统调用的清单。例如，如果只希望查看系统调用getuid和getgid，就可以使用如下的格式：

```
$ strace -e trace=getpid,getgid ./syscalltest2
getpid()                                = 7799
getgid()                                = 501
$
```

### 12.5.3　监视程序系统调用

strace程序最好的特性之一是它可以用于系统上的任何程序（假设具有执行此程序的权限）。为了使strace程序工作，不必使用专门的特性对程序进行汇编或者编译。

这可以在处理出现问题的程序时提供丰富的信息。在只关心应用程序如何执行特定任务时，使用它也很方便。

在前一节中，使用id命令显示正在运行的程序的当前用户和组ID。我们来仔细分析这个命令的strace输出。首先，按照一般模式使用strace监视id命令：

```
$ strace -o outfile id
```

这个命令创建outfile文件，它包含id命令生成的所有系统调用。可以使用编辑器查看outfile文件。在我的系统上创建的这个文件包含223行系统调用——我们必须分析的系统调用有这么多！为了帮助组织这些调用，尝试使用-c参数，这会按照时间排列调用：

```
$ strace -c id
execve("/usr/bin/id", ["id"], [/* 38 vars */]) = 0
uid=501(rich) gid=501(rich)
groups=501(rich),22(cdrom),43(usb),80(cdwriter),81(audio),503(xgrp)
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 33.98    0.000713          19        37         3 open
 14.68    0.000308          15        21           read
 14.25    0.000299           8        37           old_mmap
  7.29    0.000153          12        13           munmap
  7.24    0.000152           4        36           close
  5.77    0.000121           3        35           fstat64
  3.86    0.000081          81         1           write
  3.05    0.000064          32         2         2 connect
  2.72    0.000057          29         2           socket
  2.10    0.000044           9         5           mprotect
  2.05    0.000043           3        16           shmat
  1.33    0.000028           4         7           brk
  0.43    0.000009           9         1           ioctl
  0.38    0.000008           8         1           uname
  0.29    0.000006           3         2           ipc_subcall
  0.14    0.000003           3         1           getpid
  0.14    0.000003           3         1           semop
  0.10    0.000002           2         1           SYS_199
```

```
  0.10    0.000002           2         1           ipc_subcall
  0.10    0.000002           2         1           semget
------ ----------- ----------- --------- --------- ----------------
100.00    0.002098                   221         5 total
$
```

现在阅读和分析就容易多了。这个报告显示id程序为了执行其任务而发出的所有不同的系统调用。注意在Errors列中，系统调用open出现了3个错误，系统调用connect出现了2个错误。为了进一步进行研究，可以在strace中挑选出这些调用：

```
$ strace -e trace=open,connect id
open("/etc/ld.so.preload", O_RDONLY)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)        = 3
open("/lib/libpam.so.0", O_RDONLY)        = 3
open("/lib/libpam_misc.so.0", O_RDONLY)   = 3
open("/lib/libc.so.6", O_RDONLY)          = 3
open("/lib/libc.so.6", O_RDONLY)          = 3
open("/lib/libdl.so.2", O_RDONLY)         = 3
open("/lib/libc.so.6", O_RDONLY)          = 3
open("/lib/libdl.so.2", O_RDONLY)         = 3
open("/lib/libc.so.6", O_RDONLY)          = 3
open("/usr/share/locale/locale.alias", O_RDONLY) = 3
open("/usr/share/locale/en/LC_IDENTIFICATION", O_RDONLY) = 3
open("/usr/share/locale/en/LC_MEASUREMENT", O_RDONLY) = 3
open("/usr/share/locale/en/LC_TELEPHONE", O_RDONLY) = 3
open("/usr/share/locale/en/LC_ADDRESS", O_RDONLY) = 3
open("/usr/share/locale/en/LC_NAME", O_RDONLY) = 3
open("/usr/share/locale/en/LC_PAPER", O_RDONLY) = 3
open("/usr/share/locale/en_US/LC_MESSAGES", O_RDONLY) = 3
open("/usr/share/locale/en_US/LC_MESSAGES/SYS_LC_MESSAGES", O_RDONLY) = 3
open("/usr/share/locale/en_US/LC_MONETARY", O_RDONLY) = 3
open("/usr/share/locale/en_US/LC_COLLATE", O_RDONLY) = 3
open("/usr/share/locale/en_US/LC_TIME", O_RDONLY) = 3
open("/usr/share/locale/en_US/LC_NUMERIC", O_RDONLY) = 3
open("/usr/share/locale/en_US/LC_CTYPE", O_RDONLY) = 3
connect(3, {sin_family=AF_UNIX, path="
/var/run/.nscd_socket"}, 110) = -1 ENOENT (No such file or directory)
open("/etc/nsswitch.conf", O_RDONLY)      = 3
open("/etc/ld.so.cache", O_RDONLY)        = 3
open("/lib/libnss_files.so.2", O_RDONLY)  = 3
open("/etc/passwd", O_RDONLY)             = 3
connect(3, {sin_family=AF_UNIX, path="
/var/run/.nscd_socket"}, 110) = -1 ENOENT (No such file or directory)
open("/etc/group", O_RDONLY)              = 3
open("/usr/share/locale/en_US/LC_MESSAGES/sh-utils.mo", O_RDONLY) = -1 ENOENT (No
such file or directory)
open("/usr/share/locale/en/LC_MESSAGES/sh-utils.mo", O_RDONLY) = -1 ENOENT (No such
file or directory)
open("/etc/group", O_RDONLY)              = 3
open("/etc/group", O_RDONLY)              = 3
open("/etc/group", O_RDONLY)              = 3
open("/etc/group", O_RDONLY)              = 3
open("/etc/group", O_RDONLY)              = 3
open("/etc/group", O_RDONLY)              = 3
uid=501(rich) gid=501(rich)
groups=501(rich),22(cdrom),43(usb),80(cdwriter),81(audio),503(xgrp)
$
```

能够从这个报告发现，在3个实例中系统调用open打开库文件失败，并且系统调用connect两次连接本地套接字失败。

### 12.5.4　附加到正在运行的程序

strace程序的另一个非常好的特性是监视已经运行在系统上的程序的能力。-p参数可以把strace程序附加到一个PID并且捕获系统调用。

为了演示这个特性，我们需要一个可以在后台模式下运行的程序，并且这个程序将维持运行一段时间。nanotest.s程序用来完成这个工作：

```
# nanotest.s - Another example of using system calls
.section .data
timespec:
   .int 5, 0
output:
   .ascii "This is a test\n"
output_end:
   .equ len, output_end - output
.section .bss
   .lcomm rem, 8
.section .text
.globl _start
_start:
   nop
   movl $10, %ecx
loop1:
   pushl %ecx
   movl $4, %eax
   movl $1, %ebx
   movl $output, %ecx
   movl $len, %edx
   int $0x80

   movl $162, %eax
   movl $timespec, %ebx
   movl $rem, %ecx
   int $0x80
   popl %ecx
   loop loop1

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

nanotest.s程序使用系统调用nanosleep（系统调用值162），程序在两次把文本消息显示到标准输出（使用系统调用write）之间休眠5秒钟。系统调用nanosleep是另一个使用难以处理的数据类型作为输入值的系统调用。man页中它的提要如下：

```
SYNOPSIS
       #include <time.h>

       int  nanosleep(const struct timespec *req, struct timespec *rem);
```

第一个输入参数是系统调用在释放之前等待的时间。第二个输入参数指定一个位置，如果

系统调用在设置时间到期之前被中断，那么在这个位置中存储剩余的时间。这两个输入值都使用timespec结构保存时间值。nanosleep的man页很好地指出了timespec结构的格式：

```
struct timespec
{
        time_t  tv_sec;         /* seconds */
        long    tv_nsec;        /* nanoseconds */
};
```

这个结构使用两个值。第一个值指定时间的秒部分（使用数据类型time_t），第二个值指定时间的纳秒部分（使用长整数数据类型）。在汇编语言程序中，这两个值都可以用32位整数值来表示。使用.int命令在内存中创建timespec结构，带有两个值：

```
timespec:
   .int 5, 0
```

第一个整数值指定计时器将等待的秒数，第二个整数值指定纳秒数。这一配置将系统调用nanosleep设置为等待5秒钟。内存位置rem用于保存系统调用nanosleep的任何返回值，如果在完成之前应该中断它的话。

LOOP指令用于在程序中循环10次（循环值存储在ECX寄存器中）。因为系统调用write和nanosleep会改变ECX寄存器的值，所以在执行系统调用之前必须把这个值压入堆栈，调用完成后弹出堆栈。

汇编和连接程序之后，就可以跟踪它了。如果能够在Linux系统上同时打开两个终端会话，可以在其中一个会话中正常地运行nanotest程序，在另一个会话中运行strace程序。如果无法同时运行两个终端会话，可以使用&符号在后台模式下运行程序，就像这样：

```
$ ./nanotest &
[1] 4181
$
```

程序启动，并且立即返回到命令提示符，准备运行另一个命令。nanotest程序的输出仍然会出现在屏幕上，所以键入的时候要小心。

在后台运行程序时，会方便地给出要跟踪的正在运行的程序的PID数字。如果在单独的终端运行程序，就需要使用grep命令手动地确定进程的PID：

```
$ ps ax | grep nanotest
 4181 pts/0    S      0:00 ./nanotest
$
```

根据ps命令的显示，正在运行的进程的PID是4181。知道PID之后，就可以把strace程序附加到正在运行的nanotest程序，并且监视系统调用：

```
$ strace -p 4181
write(1, "This is a test\n", 15)        = 15
nanosleep({5, 0}, {0, 140000000})       = 0
write(1, "This is a test\n", 15)        = 15
nanosleep({5, 0}, {0, 140000000})       = 0
write(1, "This is a test\n", 15)        = 15
nanosleep({5, 0}, {0, 140000000})       = 0
```

```
write(1, "This is a test\n", 15)        = 15
nanosleep({5, 0}, {0, 140000000})       = 0
write(1, "This is a test\n", 15)        = 15
nanosleep({5, 0}, {0, 140000000})       = 0
write(1, "This is a test\n", 15)        = 15
nanosleep({5, 0}, {0, 140000000})       = 0
write(1, "This is a test\n", 15)        = 15
nanosleep({5, 0}, {0, 140000000})       = 0
write(1, "This is a test\n", 15)        = 15
nanosleep({5, 0},
```

可以看到对系统调用write和nanosleep的调用，以及它们生成的返回值。

只能附加到有权限进行附加的正在运行的进程。如果正在运行的进程是作为根进程，就必须具有root权限才能进行附加。

## 12.6 系统调用和C库

从分散在本书内的例子中，可以了解到在Linux系统上有利用现有函数的另一种方法。C库提供了丰富的函数，可以在汇编语言程序中利用它们。本节介绍如何获得系统上可用的C库函数的信息以及如何使用它们。然后是对C库和系统调用的简短比较。

### 12.6.1 C库

可以看到，C库函数为程序员提供很多有用的功能。函数包含在libc库中，必须把它连接到汇编语言程序中（参见第4章）。

C库函数的文档在man页的第3部分。和系统调用一样，C库函数的man页介绍函数的格式：

```
$ man 3 exit
NAME
       exit - cause normal program termination

SYNOPSIS
       #include <stdlib.h>

       void exit(int status);

DESCRIPTION
       The exit() function causes normal program termination and the value
       of status & 0377 is returned to the parent (see wait(2)). All func-
       tions registered with atexit() and on_exit() are called in the reverse
       order of their registration, and all open streams are flushed and
       closed. Files created by tmpfile() are removed.

       The C standard specifies two defines EXIT_SUCCESS and EXIT_FAILURE that
       may be passed to exit() to indicate successful or unsuccessful termina-
       tion, respectively.

RETURN VALUE
       The exit() function does not return.
```

与对系统调用所做的处理一样，作为汇编语言程序员，必须把C格式的函数定义解释为汇编语言格式。函数exit的定义需要单一输入参数，返回的是整数值。

第11章“使用函数”中讨论过，C样式的函数使用堆栈传递输入值。C库函数也是如此。所有输入参数都存放在堆栈中，顺序和函数提要中的顺序相反。本书中使用过的典型例子是调用C函数printf，就像这样：

```
printf("The answer is %d\n", k);
```

汇编版本是这样的：

```
pushl k
pushl $output
call printf
addl $8, %esp
```

最后一行用于复位堆栈指针以便清除堆栈中的输入值。

### 12.6.2 跟踪C函数

C库函数也是通过底层系统调用完成它们的工作。也可以使用strace程序监视C函数的执行情况。cfunctest.s程序是一个例子，它的功能与nanotest.s程序相同，但是使用C函数调用：

```
# cfunctest.s - An example of using C functions */
.section .data
output:
   .asciz "This is a test\n"
.section .text
.globl _start
_start:
   movl $10, %ecx
loop1:
   pushl %ecx
   pushl $output
   call printf
   addl $4, %esp
   pushl $5
   call sleep
   addl $4, %esp
   popl %ecx
   loop loop1
   pushl $0
   call exit
```

和nanotest.s程序类似，ECX寄存器用作循环计数器。调用printf和sleep函数之前，必须把ECX寄存器的值压入堆栈，因为这些函数的执行过程中会破坏它的值。

记住，为了创建可执行程序，必须把cfunctest.s程序和Linux系统上的C库连接在一起，并且使用动态加载器：

```
$ as -o cfunctest.o cfunctest.s
$ ld -dynamic-linker /lib/ld-linux.so.2 -lc -o cfunctest cfunctest.o
$
```

创建可执行文件之后，可以在strace程序内运行它。读者将注意到的第一件事情是它包含的系统调用比nanotest.s版本要多很多。下面只是首先发出的系统调用：

```
$ strace ./cfunctest
```

```
execve("./cfunctest", ["./cfunctest"], [/* 17 vars */]) = 0
uname({sys="Linux", node="elijah", ...}) = 0
brk(0)                                  = 0x80492e4
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40017000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=57406, ...}) = 0
old_mmap(NULL, 57406, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40018000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/libc.so.6", O_RDONLY)        = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`^\1\000"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1243076, ...}) = 0
old_mmap(NULL, 1253316, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40027000
old_mmap(0x4014e000, 36864, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3,
0x126000) = 0x4014e000
old_mmap(0x40157000, 8132, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40157000
close(3)                                = 0
munmap(0x40018000, 57406)               = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x40018000
write(1, "This is a test\n", 15This is a test
)       = 15
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({5, 0},
```

可以看到，使用C库函数时发出的系统调用比直接使用系统调用时要多很多。首先执行的几个系统调用把系统动态加载器加载到内存中（注意系统调用open打开库文件/lib/libc.so.6）。注意C函数printf对文件描述符1（STDOUT）使用相同的系统调用write，就像nanotest.s程序所做的那样。

可以使用strace参数-c查看C函数发出的所有系统调用：

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 89.67    0.001146         115        10           write
  2.82    0.000036           2        20           rt_sigprocmask
  1.64    0.000021           2        10           rt_sigaction
  1.56    0.000020           3         6           old_mmap
  1.02    0.000013           4         3         1 open
  0.86    0.000011           1        10           nanosleep
  0.63    0.000008           4         2         2 access
  0.47    0.000006           2         3           fstat64
  0.39    0.000005           5         1           munmap
  0.31    0.000004           2         2           close
  0.23    0.000003           3         1           read
  0.23    0.000003           3         1           uname
  0.16    0.000002           2         1           brk
------ ----------- ----------- --------- --------- ----------------
100.00    0.001278                    70         3 total
```

### 12.6.3 系统调用和C库的比较

虽然在汇编语言程序中C库函数需要的系统调用比直接使用系统调用时要多，但是不太可能注意到性能上有太多区别，如果对nanotest.s程序使用strace选项-c，将得到下面这样的结果：

```
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 90.71    0.000781          78        10           write
  9.29    0.000080           8        10           nanosleep
------ ----------- ----------- --------- --------- ----------------
100.00    0.000861                    20           total
$
```

注意，nanotest程序中系统调用的总时间是0.000861秒，而cfunctest程序中系统调用的总时间是0.001278秒。显然，存在时间差异的主要原因是由于C函数在执行程序之前必须加载动态加载器程序。在大型汇编语言程序中，这种开销将不这么明显。

使用原始Linux系统调用的主要原因如下：

- 它创建长度尽可能最短的代码，因为不需要把外部库连接到程序中。
- 它创建尽可能最快的代码，同样因为不需要把外部库连接到程序中。
- 连接后的可执行文件独立于任何外部库代码。

在汇编语言程序中使用C库函数的主要原因如下：

- C库包含很多函数，模拟它们需要许多汇编语言代码（比如ASCII到整数或者浮点数据类型的转换）。
- C库在操作系统之间是可移植的（比如在Intel平台上运行的FreeBSD上编译的程序也可以运行在Linux系统上）。
- C库函数可以在程序之间利用共享库，减少内存需求。

显然，使用哪种类型的函数都有其原因。基本原则是哪种方法更加适合现有应用程序，以及程序员更喜欢使用哪种方法。如果正在编写和C或者C++程序进行交互的汇编语言程序，也许读者已经习惯于使用C库函数，并且知道它们是可以使用的。

## 12.7 小结

本章讨论汇编语言程序员可以利用的Linux系统调用。Linux内核提供系统调用，系统调用进而提供对内核控制的资源的访问。Linux内核负责管理和提供程序对内存、系统上的设备、系统上的文件系统和系统上正在运行的进程的访问。

可以利用的内核系统调用有很多。/usr/include/asm/unistd.h文件列出了所有系统调用，还列出了每个系统调用的系统调用值，可以在汇编语言程序中使用这些值访问系统调用。标准Linux的man页的第2部分中提供系统调用的定义。

阅读系统调用定义时，必须确定系统调用需要什么输入值，以及必须处理什么返回值（或者多个返回值）。内核系统调用使用基于寄存器的方法传递输入值和输出值。输入值按照man页中定义的顺序传递给系统调用。使用的寄存器的顺序是EBX、ECX、EDX、ESI和EDI。这允许使用最多5个输入值。输出值返回到EAX寄存器中。

使用很多系统调用的程序常常难以查找错误，因为控制权被传递给内核以便处理系统调用。可以使用strace程序，它是跟踪使用了什么系统调用以及系统调用的结果的一种方法。strace程序在程序运行的时候监视它，并且显示发出的每个系统调用、提供给系统调用的输入值以及系统调用生成的输出值。strace程序不仅可以启动新的程序以便跟踪它，也可以附加到已经在运行的现有程序。

虽然很多汇编语言程序员利用内核系统调用提供对内存、设备和进程信息的低级访问，但是还有获得这种信息的其他途径。C库函数提供访问内核特性的中间途径。C库函数是标准函数，可以在不同系统上使用它们提供对功能的标准访问。汇编语言程序可以通过连接到系统上的C库来访问C库函数。很多程序员喜欢使用C库函数，特别是需要很多额外的汇编语言编码才能实现的比较高级的C库函数。使用C库函数通常会带来很小的开销，但是在应用程序中的收益通常胜过开销的增加。

下一章重点讲解在C和C++高级语言中使用汇编语言。这一章讨论内联汇编语言程序设计，这是在较大型C和C++应用程序内创建小型汇编语言例程的方法。很多程序员使用内联汇编语言程序设计，这要么是为了提高复杂处理的性能，要么是为了执行C编译器不能执行的任务。

# 第三部分　高级汇编语言技术

## 第13章　使用内联汇编

既然读者了解了汇编语言程序设计的基础知识，现在是开始把这些概念投入实际应用的时候了。利用汇编语言程序设计的一种非常常见的方式是在高级语言（比如C和C++）程序内编写汇编函数。完成这一工作有几种不同的方式。本章介绍如何把汇编语言函数直接放到C和C++语言程序内。这种技术称为内联汇编（inline assembly）。

本章首先介绍C和C++程序如何使用函数，以及如何使用编译器把函数转换为汇编语言代码。接下来，讨论内联汇编的基本格式，包括如何整合简单的汇编函数。之后，介绍扩展的内联汇编格式。这种格式可以在C或者C++程序内整合更加复杂的汇编语言函数。最后，本章讲解如何使用复杂的内联汇编语言函数在C程序内定义宏。

### 13.1　什么是内联汇编

在标准的C或者C++程序中，在文本源代码文件中按照C或者C++语法输入代码。然后使用编译器把源代码文件编译为汇编语言代码。这个步骤之后，汇编语言代码和所有必须的库连接在一起生成可执行程序（参见第3章“相关的工具”）。

在Linux领域中，使用GNU编译器（gcc）从文本源代码文件创建可执行程序。通常，把代码转换为汇编语言的步骤对程序员是隐藏的。但是第3章介绍过，可以使用GNU编译器的-S选项查看从源代码生成的实际汇编语言代码。

在C和C++程序设计中，常用的程序设计技术是在源代码文件中创建单独的独立函数。这些函数执行独立的处理，可以从主程序多次调用它们。当C或者C++程序被分隔为函数时，编译器把每个函数编译为独立的汇编函数（参见第11章“使用函数”）。函数仍然被包含在相同的汇编语言文件内，但是作为独立的函数。为了查看生成的内容，仍然可以使用-S选项对程序进行编译并且查看生成的汇编语言代码。

为了演示这种情况，cfunctest.c程序在一个简单的C语言程序内使用几个单独的函数：

```
/* cfunctest.c - An example of functions in C */
#include <stdio.h>

float circumf(int a)
{
        return 2 * a * 3.14159;
}

float area(int a)
{
```

```
        return a * a * 3.14159;
}

int main()
{
        int x = 10;
        printf("Radius: %d\n", x);
        printf("Circumference: %f\n", circumf(x));
        printf("Area: %f\n",area(x));
        return 0;
}
```

这个程序定义了两个函数，它们使用单一整数值作为输入，并且生成双精度浮点值作为输出。数学运算在独立的函数中执行，和主程序代码分开。在主程序中可以按照需要多次调用函数，不必编写额外的代码。

为了查看编译器生成的汇编语言代码，使用-S选项进行编译。

```
$ gcc -S cfunctest.c
```

这个命令创建文件cfunctest.s，如下所示：

```
.file   "cfunctest.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
        .align 8
.LC0:
        .long   0xf01b866e,0x400921f9
.text
        .align 16
.globl circumf
        .type    circumf,@function
circumf:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    8(%ebp), %eax
        addl    %eax, %eax
        pushl   %eax
        fildl   (%esp)
        popl    %eax
        fldl    .LC0
        fmulp   %st, %st(1)
        fstps   -4(%ebp)
        flds    -4(%ebp)
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe1:
        .size    circumf,.Lfe1-circumf
                .section        .rodata
        .align 8
.LC2:
        .long   0xf01b866e,0x400921f9
.text
        .align 16
```

```
.globl area
        .type    area,@function
area:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    8(%ebp), %eax
        imull   8(%ebp), %eax
        pushl   %eax
        fildl   (%esp)
        popl    %eax
        fldl    .LC2
        fmulp   %st, %st(1)
        fstps   -4(%ebp)
        flds    -4(%ebp)
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe2:
        .size    area,.Lfe2-area
                .section        .rodata
.LC4:
        .string "Radius: %d\n"
.LC5:
        .string "Circumference: %f\n"
.LC6:
        .string "Area: %f\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $10, -4(%ebp)
        subl    $8, %esp
        pushl   -4(%ebp)
        pushl   $.LC4
        call    printf
        addl    $16, %esp
        subl    $4, %esp
        subl    $8, %esp
        pushl   -4(%ebp)
        call    circumf
        addl    $12, %esp
        leal    -8(%esp), %esp
        fstpl   (%esp)
        pushl   $.LC5
        call    printf
        addl    $16, %esp
        subl    $4, %esp
        subl    $8, %esp
        pushl   -4(%ebp)
        call    area
        addl    $12, %esp
        leal    -8(%esp), %esp
```

```
        fstpl   (%esp)
        pushl   $.LC6
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe3:
        .size    main,.Lfe3-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

现在读者应该能够理解编译器生成的汇编语言代码。两个C函数被创建为单独的汇编语言函数，和主程序代码分开。主程序使用标准的C样式函数格式把输入参数传递给函数（通过把输入值存放在堆栈的顶部）。CALL指令用于从主程序调用函数。

在这个简单的例子中，生成的汇编代码执行的功能是非常微不足道的。但是，在比较复杂的应用程序中，也许不希望编译器生成汇编语言代码，或者也许希望使用编译器不可能生成的汇编语言指令（比如CPUID指令）。

如果想要直接控制生成什么汇编语言代码去实现函数，可以有如下3种选择：

- 从头开始编写汇编语言代码来实现函数，然后从C程序调用它。
- 使用-S选项创建C代码的汇编语言版本，在必要的情况下修改汇编语言代码，然后连接汇编语言代码从而生成可执行文件。
- 在原始的C代码内创建函数的汇编语言代码，然后使用标准C编译器进行编译。

第14章“调用汇编库”中讨论第一种选择。第15章“优化例程”中讨论第二种选择。第三种选项恰恰就是内联汇编语言程序设计的工作方式。这种方法可以在C或者C++源代码本身之内创建汇编语言函数，不必连接额外的库或者程序。这种方法对于最终程序在汇编语言级别如何实现特定的函数，给予程序员更多的控制权。

## 13.2 基本的内联汇编代码

创建内联汇编代码和创建汇编函数没有太大区别，除了这是在C或者C++程序内完成的之外。本节介绍如何创建基本的内联汇编代码函数，它们可以在C或者C++程序内实现简单的汇编语言代码。

### 13.2.1 asm格式

GNU的C编译器使用asm关键字指出使用汇编语言编写的源代码段落。asm段的基本格式如下：

```
asm( "assembly code" );
```

包含在括号中的汇编代码必须按照特定的格式：

- 指令必须括在引号里。
- 如果包含的指令超过一条，那么必须使用新行字符分隔汇编语言代码的每一行。通常，还包含制表符帮助缩进汇编语言代码，使代码行更容易阅读。

需要第二个规则是因为编译器逐字地取得asm段中的汇编代码，并且把它们放在为程序生成的汇编代码中。每条汇编语言指令都必须在单独的一行中——因此需要包含新行字符。

一些汇编器还要求使用制表符字符缩进指令以便它们和标签区别开。GNU编译器不需要这样做，但是很多程序员使用制表符字符以便保持一致性。

这些要求可能导致在源代码中创建出看上去有些令人混淆的汇编语言代码，但是有助于使生成的汇编语言代码更合理。

下面是基本内联汇编段的一个例子：

```
asm ("movl $1, %eax\n\tmovl $0, %ebx\n\tint $0x80");
```

这个例子使用3条指令：两个MOVL指令，把值1存放到EAX寄存器中，把值0存放到EBX寄存器中，还有执行Linux系统调用的INT指令。

在使用很多汇编指令时，这种格式显得有些凌乱。大多数程序员把指令放在单独的行中。这样做的时候，每条指令都必须括在引号里面：

```
asm ( "movl $1, %eax\n\t"
      "movl $0, %ebx\n\t"
      "int $0x80");
```

这种格式就容易阅读了，在调试应用程序时会轻松得多。asm段可以被放在C或C++源代码中的任何地方。下面的asmtest.c程序演示asm段在实际的程序中是什么样子：

```
/* asmtest.c - An example of using an asm section in a program*/
#include <stdio.h>

int main()
{
   int a = 10;
   int b = 20;
   int result;
   result = a * b;
   asm ( "nop");
   printf("The result is %d\n", result);
   return 0;
}
```

在这个C程序中，asm语句里面的汇编语言指令（NOP指令）不执行什么任务，但是它会出现在编译器生成的汇编语言代码中。为了生成这个程序的汇编语言代码，需要使用gcc命令的-S选项。生成的汇编代码文件应该如下：

```
.file   "asmtest.c"
        .section        .rodata
.LC0:
        .string "The result is %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $10, -4(%ebp)
        movl    $20, -8(%ebp)
```

```
        movl    -4(%ebp), %eax
        imull   -8(%ebp), %eax
        movl    %eax, -12(%ebp)
#APP
        nop

#NO_APP
        movl    -12(%ebp), %eax
              movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.2 (Debian)"
```

生成的代码使用一般的C样式函数开头并且使用LEAVE指令实现标准的结尾（参见第11章）。在开头和结尾代码之内是C源代码生成的代码，其中有一个由#APP和#NO_APP符号标识的段落。这个段落包含asm段指定的内联汇编代码。注意如何使用新行和制表符字符指定代码的布局。

### 13.2.2 使用全局C变量

仅仅实现汇编语言代码本身并不能完成很多任务。为了完成任何实际的工作，必须有把数据传递进和传递出内联汇编语言函数的途径。

基本的内联汇编代码可以利用应用程序中定义的全局C变量。这里要记住的是“全局”这个词。只有全局定义的变量才能在基本的内联汇编代码内使用。通过C程序中使用的相同名称引用这种变量。

globaltest.c程序演示如何进行这样的操作：

```
/* globaltest.c - An example of using C variables */
#include <stdio.h>

int a = 10;
int b = 20;
int result;

int main()
{
      asm ( "pusha\n\t"
            "movl a, %eax\n\t"
            "movl b, %ebx\n\t"
            "imull %ebx, %eax\n\t"
            "movl %eax, result\n\t"
            "popa");
      printf("the answer is %d\n", result);
      return 0;
}
```

在这个C程序中，变量a、b和result被定义为全局变量，并且在代码的asm段中使用。注意，在汇编语言代码中值被用作内存位置，而不是立即数值。在C程序的任何其他位置也可以按照一

般的方式使用这种变量。

记住，数据变量必须被声明为全局的。不能在asm段中使用局部变量。

编译器生成的汇编代码是这样的:

```
.file   "globaltest.c"
.globl a
        .data
        .align 4
        .type   a, @object
        .size   a, 4
a:
        .long   10
.globl b
        .align 4
        .type   b, @object
        .size   b, 4
b:
        .long   20
        .section        .rodata
.LC0:
        .string "The result is %d\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
#APP
        pusha
        movl a, %eax
        movl b, %ebx
        imull %ebx, %eax
        movl %eax, result
        popa
#NO_APP
        movl    result, %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .comm   result,4,4
        .section        .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU) 3.3.2 (Debian)"
```

注意在.data段中如何声明变量a和b并且为它们赋予适当的值。因为C代码中没有对变量result进行初始化，所以把它声明为.comm值。

这个范例程序中显示了另一个重要特性。注意汇编语言代码开头的PUSHA指令，以及结尾的POPA指令。在进入代码之前存储寄存器的初始值，然后在完成代码之后恢复它们，记住这一

点很重要。在编译后的C源代码中，编译器很有可能会使用这些寄存器存放其他值。如果在asm段中修改它们，可能会发生不可预料的后果。

### 13.2.3 使用volatile修饰符

在应用程序中创建内联汇编代码时，必须了解编译器在编译操作的过程中可能进行的操作。在一般的C或者C++应用程序中，编译器也许会试图优化生成的汇编代码以提高性能。通常这是通过这样的方式完成的：消除不使用的函数，在不同时使用的值之间共享寄存器，以及重新编排代码以便实现更好的程序流程。

对于内联汇编函数来说，有时候优化并不是好事情。编译器也可能查看内联代码并且试图优化它，这可能会产生不希望的后果。

如果希望编译器不处理手动编码的内联汇编函数，可以明确地这么说明！可以把volatile修饰符放在asm语句中表示不希望优化这个代码段。使用volatile修饰符的asm语句的格式如下：

```
asm volatile ("assembly code");
```

这个语句内的汇编代码使用不带volatile修饰符时使用的标准规则。volatile修饰符的添加不改变在内联汇编代码中存储和获得寄存器值的要求。

### 13.2.4 使用替换的关键字

如果必须的话，可以改变用于标识内联汇编代码段的关键字asm。ANSI C规范把关键字asm用于其他用途，不能将它用于内联汇编语句。如果使用ANSI C约定编写代码，你必须使用关键字__asm__替换一般的关键字asm。

语句中的汇编代码段不必改动，和使用关键字asm时一样，就像下面的例子：

```
__asm__ ("pusha\n\t"
        "movl a, %eax\n\t"
        "movl b, %ebx\n\t"
        "imull %ebx, %eax\n\t"
        "movl %eax, result\n\t"
        "popa");
```

关键字__asm__也可以使用修饰符__volatile__进行修饰。

## 13.3 扩展asm

基本的asm格式提供创建汇编代码的简单方式，但是有其局限性。首先，所有输入值和输出值都必须使用C程序的全局变量。其次，必须极为注意在内联汇编代码中不去改变任何寄存器的值。

GNU编译器提供asm段的扩展格式来帮助解决这些问题。扩展格式提供附加的选项，可以更加精确地控制在C或者C++语言程序中如何生成内联汇编语言代码。本节介绍扩展asm格式。

### 13.3.1 扩展asm格式

因为扩展asm格式提供附加的特性，所以它们必须采用新的格式。asm扩展版本的格式如下：

```
asm ("assembly code" : output locations : input operands : chagned registers);
```

这种格式由4个部分构成，使用冒号分隔：

- 汇编代码：使用和基本asm格式相同的语法的内联汇编代码
- 输出位置：包含内联汇编代码的输出值的寄存器和内存位置的列表
- 输入操作数：包含内联汇编代码的输入值的寄存器和内存位置的列表
- 改动的寄存器：内联代码改变的任何其他寄存器的列表

在扩展asm格式中，并不是所有这些部分都必须出现。如果汇编代码不生成输出值，这个部分就必须为空，但是必须使用两个冒号把汇编代码和输入操作数分隔开。如果内联汇编代码不改动寄存器的值，那么可以忽略最后的冒号。

下面几节介绍如何使用扩展asm格式。

### 13.3.2 指定输入值和输出值

在基本asm格式中，在汇编代码中通过C全局变量名称整合输入值和输出值。使用扩展格式时的方法稍有不同。

在扩展格式中，可以从寄存器和内存位置给输入值和输出值赋值。输入值和输出值列表的格式是：

```
"constraint" (variable)
```

其中variable是程序中声明的C变量。在扩展asm格式中，局部和全局变量都可以使用。constraint定义把变量存放到哪里（对于输入值）或者从哪里传送变量（对于输出值）。使用它定义把变量存放在寄存器中还是内存位置中。

约束是单一字符的代码。约束代码如下表所示。

| 约 束 | 描 述 |
|---|---|
| a | 使用%eax、%ax或者%al寄存器 |
| b | 使用%ebx、%bx或者%bl寄存器 |
| c | 使用%ecx、%cx或者%cl寄存器 |
| d | 使用%edx、%dx或者%dl寄存器 |
| S | 使用%esi或者%si寄存器 |
| D | 使用%edi或者%di寄存器 |
| r | 使用任何可用的通用寄存器 |
| q | 使用%eax、%ebx、%ecx或者%edx寄存器之一 |
| A | 对于64位值使用%eax和%edx寄存器 |
| f | 使用浮点寄存器 |
| t | 使用第一个（顶部的）浮点寄存器 |
| u | 使用第二个浮点寄存器 |
| m | 使用变量的内存位置 |
| o | 使用偏移内存位置 |
| V | 只使用直接内存位置 |
| i | 使用立即整数值 |
| n | 使用值已知的立即整数值 |
| g | 使用任何可用的寄存器或者内存位置 |

除了这些约束之外，输出值还包含一个约束修饰符，它指示编译器如何处理输出值。可以使用的输出修饰符如下表所示。

| 输出修饰符 | 描　述 |
|---|---|
| + | 可以读取和写入操作数 |
| = | 只能写入操作数 |
| % | 如果必要，操作数可以和下一个操作数切换 |
| & | 在内联函数完成之前，可以删除或者重新使用操作数 |

了解输入值和输出值如何工作的最简单的途径是查看一些例子。下面这个例子:

```
asm ("assembly code" : "=a"(result) : "d"(data1), "c"(data2));
```

把C变量data1存放在EDX寄存器中，把data2存放到ECX寄存器中。内联汇编代码的结果将存放在EAX寄存器中，然后传送给变量result。

### 13.3.3 使用寄存器

如果输入值和输出变量被赋值给寄存器，那么在内联汇编代码中几乎可以像平常一样使用寄存器。我使用“几乎”这个词是因为有个特殊情况需要处理。

在扩展asm格式中，为了在汇编代码中引用寄存器，必须使用两个百分号符号，而不是一个(其原因将在稍后讨论)。这使代码看上去稍显奇怪，但是没有太多区别。

regtest1.c程序演示在扩展asm格式中使用寄存器:

```
/* regtest1.c - An example of using registers */
#include <stdio.h>

int main()
{
   int data1 = 10;
   int data2 = 20;
   int result;

   asm ("imull %%edx, %%ecx\n\t"
        "movl %%ecx, %%eax"
        : "=a"(result)
        : "d"(data1), "c"(data2));

   printf("The result is %d\n", result);
   return 0;
}
```

这一次，C变量被声明为局部变量，使用基本asm格式时不能使用它。每个C变量被赋值给特定的寄存器。使用等号符号修饰输出寄存器表明汇编代码只能写入它（这是对内联代码中所有输出值的要求)。

编译这个C程序时，编译器自动生成实现内联汇编代码所需的把C变量存放到正确寄存器的汇编代码。可以再次使用-S选项查看生成的代码。生成的内联代码如下:

```
        movl    $10, -4(%ebp)
        movl    $20, -8(%ebp)
```

```
        movl    -4(%ebp), %edx
        movl    -8(%ebp), %ecx
#APP
        imull %edx, %ecx
        movl %ecx, %eax
#NO_APP
        movl    %eax, -12(%ebp)
```

编译器把data1和data2的值传送到为C变量保留的堆栈空间。然后按照内联汇编代码的要求，把这些值加载到EDX和ECX寄存器中。然后把EAX寄存器中生成的结果输出传送到堆栈中的变量位置result。

不一定总要在内联汇编段中指定输出值。一些汇编指令已经假设输入值包含输出值。

MOVS指令在输入值内包含输出位置。movstest.c程序演示这种情况：

```
/* movstest.s - An example of instructions with only input values */
#include <stdio.h>

int main()
{
   char input[30] = {"This is a test message.\n"};
   char output[30];
   int length = 25;

   asm volatile ("cld\n\t"
       "rep movsb"
      :
      : "S"(input), "D"(output), "c"(length));

   printf("%s", output);
   return 0;
}
```

movstest.c程序把MOVS指令需要的3个输入值指定为输入值。要复制的字符串的位置存放在ESI寄存器中，目标位置存放在EDI寄存器中，要复制的字符串的长度存放在ECX寄存器中（记住字符串长度中包含结尾的空字符）。

输出值已经被定义为输入值之一，所以在扩展格式中没有专门定义输出值。因为没有定义专门的输出值，所以使用关键字volatile很重要；否则，编译器也许会认为这个asm段是不必要的而删除它，因为它不生成输出。

### 13.3.4 使用占位符

在regtest1.c程序例子中，输入值存放在内联汇编段中声明的特定寄存器中，并且在汇编指令中专门使用这些寄存器。虽然这种方式能够很好地处理只有几个输入值的情况，但是对于需要很多输入值的函数，这种方式就显得有些繁琐。

为了帮助解决这个问题，扩展asm格式提供了占位符（placeholder），可以在内联汇编代码中使用它引用输入值和输出值。这样可以在对于编译器方便的任何寄存器或者内存位置中声明输入和输出值。

占位符是前面加上百分号符号的数字。按照内联汇编代码中列出的每个输入值和输出值在列

表中的位置，每个值被赋予一个从零开始的数字。然后就可以在汇编代码中使用占位符表示值。

例如，下面的内联代码：

```
asm ("assembly code"
     : "=r"(result)
     : "r"(data1), "r"(data2));
```

将生成如下的占位符：

- %0将表示包含变量值result的寄存器。
- %1将表示包含变量值data1的寄存器。
- %2将表示包含变量值data2的寄存器。

注意，占位符提供在内联汇编代码中利用寄存器和内存位置的方法。汇编代码中使用占位符只作为原始的数据类型：

```
imull %1, %2
movl %2, %0
```

记住，必须把输入值和输出值声明为内联代码中汇编指令需要的正确的存储元素（寄存器或者内存）。在这个例子中，两个输入值都必须加载到寄存器中以供IMULL指令使用。

为了演示占位符的使用，regtest2.c程序执行和regtest1.c程序相同的功能，但是允许编译器选择使用哪些寄存器：

```
/* regtest2.c - An example of using placeholders */
#include <stdio.h>

int main()
{
   int data1 = 10;
   int data2 = 20;
   int result;

   asm ("imull %1, %2\n\t"
        "movl %2, %0"
        : "=r"(result)
        : "r"(data1), "r"(data2));

   printf("The result is %d\n", result);
   return 0;
}
```

当定义输入值和输出值时，regtest2.c程序使用约束r，这表示使用寄存器满足所有数据需求。生成程序的汇编语言代码时，编译器选择要使用的寄存器。为了验证这一点，可以使用-S选项查看生成的汇编代码：

```
        movl    $10, -4(%ebp)
        movl    $20, -8(%ebp)
        movl    -4(%ebp), %edx
        movl    -8(%ebp), %eax
#APP
        imull %edx, %eax
        movl %eax, %eax
#NO_APP
        movl    %eax, -12(%ebp)
```

生成汇编代码时，我的编译器选择了有趣的方式。它使用EDX寄存器保存值data1，使用EAX寄存器保存值data2，就像我们平常期望的一样。有趣的部分是它注意到在输入值被使用完之后才生成结果，所以它把结果变量也赋值给EAX寄存器。我的结构不良的内联汇编代码仍然执行MOVL指令，但只是使EAX寄存器传送给自己。

可以在调试器中监视程序的运行，以便查看MOVL指令是否确实执行了。为了生成能够在调试器中使用的可执行程序，可以使用gcc编译器的-gstabs选项:

```
$ gcc -gstabs -o regtest2 regtest2.c
```

创建好可执行文件后，可以在调试器中运行它:

```
$ gdb -q regtest2
(gdb) break *main
Breakpoint 1 at 0x8048364: file regtest2.c, line 4.
(gdb) run
Starting program: /home/rich/palp/chap13/regtest2

Breakpoint 1, main () at regtest2.c:4
4       {
(gdb) s
5               int data1 = 10;
(gdb) s
6               int data2 = 20;
(gdb) s
9               asm ("imull %1, %2\n\t"
(gdb) s
14              printf("The result is %d\n", result);
(gdb) info reg
eax            0xc8     200
ecx            0x1      1
edx            0xa      10
```

为了在C程序中设置断点，可以指定开始的行号或者函数标签。这个例子在函数标签main ()的位置（即程序的开头）设置断点。

当单步执行程序时，读者也许会注意到的一件事情是调试器认为asm段是单一语句。可以使用调试器的stepi命令进入asm段进行单步执行，单独地执行每条指令。

寄存器列表显示在asm段之后data1值被加载到EDX寄存器中，EAX寄存器被用作结果变量。

### 13.3.5 引用占位符

就像在regtest2.c程序中看到的，我不必要地使用MOVL指令在适当的变量中生成输出值。有时候使用相同的变量作为输入值和输出值是有好处的。要这样做，必须在扩展asm段中区别定义输入值和输出值。

如果内联汇编代码中的输入值和输出值共享程序中相同的C变量，可以指定使用占位符作为约束值。这样会创建看起来有些奇怪的代码，但是这便于减少代码中需要的寄存器的数量。

为了修改regtest2.c程序的内联代码，可以编写下面的代码:

```
asm ("imull %1, %0"
     : "=r"(data2)
     : "r"(data1), "0"(data2));
```

0标记通知编译器使用第一个命名的寄存器存放输出值data2。第二行代码定义第一个命名寄存器，它为输入变量data2分配一个寄存器。这样确保将使用相同的寄存器保存输入值和输出值。当然，内联代码完成时，结果将被存放到值data2中。

regtest3.c程序演示这种做法：

```
/* regtest3.c - An example of using placeholders for a common value */
#include <stdio.h>

int main()
{
   int data1 = 10;
   int data2 = 20;

   asm ("imull %1, %0"
        : "=r"(data2)
        : "r"(data1), "0"(data2));

   printf("The result is %d\n", data2);
   return 0;
}
```

regtest3.c程序使用值data2同时作为输入值和输出值。

### 13.3.6 替换的占位符

如果处理很多输入值和输出值，数字型的占位符很快就会变得混乱。为了使条理清晰，GNU编译器（从版本3.1开始）允许声明替换的名称作为占位符。

替换的名称在声明输入值和输出值的段中定义。格式如下：

```
%[name]"constraint"(variable)
```

定义的值name成为内联汇编代码中变量的新的占位符标识符，如下面的例子所示：

```
asm ("imull %[value1], %[value2]"
     : [value2] "=r"(data2)
     : [value1] "r"(data1), "0"(data2));
```

使用替换的占位符名称的方式和使用普通的占位符相同，就像下面alttest.c程序演示的：

```
/* alttest.c - An example of using alternative placeholders */
#include <stdio.h>

int main()
{
   int data1 = 10;
   int data2 = 20;

   asm ("imull %[value1], %[value2]"
        : [value2] "=r"(data2)
        : [value1] "r"(data1), "0"(data2));

   printf("The result is %d\n", data2);
   return 0;
}
```

### 13.3.7 改动的寄存器列表

在迄今为止提供的例子中，读者也许会注意到我没有在扩展asm格式中使用改动的寄存器列表，尽管每个程序显然都包含改动了的寄存器。

编译器假设输入值和输出值使用的寄存器会被改动，并且相应地做出处理。程序员不需要在改动的寄存器列表中包含这些值，如果这样做了，就会产生错误消息，如下面的badregtest.c程序所示：

```
/* badregtest.c - An example of incorrectly using the changed registers list */
#include <stdio.h>

int main()
{
   int data1 = 10;
   int result = 20;

   asm ("addl %1, %0"
        : "=d"(result)
        : "c"(data1), "0"(result)
        : "%ecx", "%edx");

   printf("The result is %d\n", result);
   return 0;
}
```

badregtest.c程序指定变量result应该加载到EDX寄存器中，变量data1加载到ECX寄存器中。改动的寄存器列表错误地指定在内联代码中改动ECX和寄存器EDX寄存器。注意改动的寄存器列表中的寄存器使用完整的寄存器名称，而不像输入和输出寄存器定义那样仅仅是单一字母。在寄存器名称前面使用百分号符号是可选的。

当试图编译这个程序时，会出现错误：

```
$ gcc -o badregtest badregtest.c
badregtest.c: In function 'main':
badregtest.c:8: error: can't find a register in class 'DREG' while reloading 'asm'
$
```

编译器已经知道EDX寄存器用作寄存器，它不能正确地处理改动的寄存器列表的要求。

改动的寄存器列表的正确使用方法是，如果内联汇编代码使用了没有被初始地声明为输入值或者输出值的任何其他寄存器，则要通知编译器。编译器必须知道这些寄存器，以便避免使用它们，如changedtest.c程序所示：

```
/* changedtest.c - An example of setting registers in the changed registers list */
#include <stdio.h>

int main()
{
   int data1 = 10;
   int result = 20;

   asm ("movl %1, %%eax\n\t"
        "addl %%eax, %0"
```

```
        : "=r"(result)
        : "r"(data1), "0"(result)
        : "%eax");

   printf("The result is %d\n", result);
   return 0;
}
```

在changedtest.c程序中，内联汇编代码使用EAX寄存器作为存储数据值的中间位置。因为这个寄存器没有被声明为输入值或者输出值，所以必须在改动的寄存器列表中包含它。

现在编译器知道EAX寄存器不可用，它会避免使用这个寄存器。使用约束r声明输入值和输出值，这使编译器能够选择要使用的寄存器。查看生成的汇编语言代码，可以发现选择了哪些寄存器：

```
        movl    $10, -4(%ebp)
        movl    $20, -8(%ebp)
        movl    -4(%ebp), %ecx
        movl    -8(%ebp), %edx
#APP
        movl %ecx, %eax
        addl %eax, %edx
#NO_APP
        movl    %edx, %eax
```

把C变量传送到寄存器中的代码使用ECX和EDX寄存器（记住在regtest2.c程序中使用的是EAX和EDX寄存器）。编译器正确地避免使用EAX寄存器，因为在内联汇编代码中声明了要使用它。

改动的寄存器列表有个奇怪的地方：如果在内联汇编代码之内使用了没有在输入值或者输出值中定义的任何内存位置，那它必须被标记为被破坏的。在改动的寄存器列表中使用“memory”这个词通知编译器这个内存位置在内联汇编代码中被改动。

### 13.3.8 使用内存位置

虽然在内联汇编语言代码中使用寄存器比较快，但是也可以直接使用C变量的内存位置。约束m用于引用输入值和输出值中的内存位置。记住，对于要求使用寄存器的汇编指令，仍然必须使用寄存器，所以也许不得不定义保存数据的中间寄存器。memtest.c程序演示这种情况：

```
/* memtest.c - An example of using memory locations as values */
#include <stdio.h>

int main()
{
   int dividend = 20;
   int divisor = 5;
   int result;

   asm("divb %2\n\t"
       "movl %%eax, %0"
       : "=m"(result)
       : "a"(dividend), "m"(divisor));

   printf("The result is %d\n", result);
   return 0;
}
```

asm段把被除数的值加载到EAX寄存器中，DIV指令需要它。除数存放在内存位置中，作为输出值。生成的汇编代码如下：

```
        movl    $20, -4(%ebp)
        movl    $5, -8(%ebp)
        movl    -4(%ebp), %eax
#APP
        divb -8(%ebp)
        movl %eax, -12(%ebp)
#NO_APP
```

值被加载到内存位置中（在堆栈中），被除数的值也被传送到EAX寄存器中。确定结果之后，它被传送到堆栈中它的内存位置中，而不是寄存器中。

因为这个例子使用DIVB指令，所以只能处理小于65 536的被除数值和小于256的除数值。如果希望使用更大的值，就必须将内联汇编语言代码改为使用DIVW或者DIVL指令。

### 13.3.9 使用浮点值

因为FPU以堆栈方式使用寄存器，所以在编写内联汇编语言代码的过程中使用浮点值就有一点儿区别。必须更加留意内联代码处理FPU寄存器的方式。

读者也许已经注意到处理FPU寄存器堆栈的约束有3个：

- f引用任何可用的浮点寄存器
- t引用顶部的浮点寄存器
- u引用第二个浮点寄存器

从FPU获得输出值的时候，不能使用约束f；必须声明约束t或者u来指定输出值所在的FPU寄存器，就像下面的例子：

```
asm( "fsincos"
     : "=t"(cosine), "=u"(sine)
     : "0"(radian));
```

FSINCOS指令把输出值存放在FPU堆栈的前两个寄存器中。必须为正确的输出值指定正确的寄存器。因为输入值也必须存放在ST(0)寄存器中，所以它和第一个输出值使用相同的寄存器，并且使用占位符声明它。sincostest.c程序演示使用这段内联汇编代码的情况：

```
/* sincostest.c - An example of using two FPU registers */
#include <stdio.h>

int main()
{
   float angle = 90;
   float radian, cosine, sine;

   radian = angle / 180 * 3.14159;

   asm("fsincos"
       :"=t"(cosine), "=u"(sine)
       :"0"(radian));

   printf("The cosine is %f, and the sine is %f\n", cosine, sine);
   return 0;
}
```

编译器生成的这个函数的汇编语言代码如下：

```
        flds    -8(%ebp)
#APP
        fsincos
#NO_APP
        fstps   -24(%ebp)
        movl    -24(%ebp), %eax
        movl    %eax, -12(%ebp)
        fstps   -24(%ebp)
        movl    -24(%ebp), %eax
        movl    %eax, -16(%ebp)
```

使用FLDS指令，从程序堆栈把radian值加载到FPU堆栈中。执行FSINCOS指令之后，使用FSTPS指令把两个输出值弹出堆栈，并且传送到它们正确的C变量位置。

在前面的例子中，因为编译器知道输出值位于前两个FPU寄存器中，所以它会弹出这些值，把FPU堆栈恢复为以前的状态。如果在FPU堆栈中执行的任何操作没有被清除，就必须在改动的寄存器列表中指定适当的FPU寄存器。areatest.c程序演示这种情况：

```
/* areatest.c - An example of using floating point regs */
#include <stdio.h>

int main()
{
   int radius = 10;
   float area;

   asm("fild %1\n\t"
       "fimul %1\n\t"
       "fldpi\n\t"
       "fmul %%st(1), %%st(0)"
       : "=t"(area)
       :"m"(radius)
       : "%st(1)");

   printf("The result is %f\n", area);
   return 0;
}
```

areatest.c程序把半径值存放在一个内存位置中，然后使用FILD指令把这个值加载到FPU堆栈的顶部。这个值和自己相乘，结果仍然在ST(0)寄存器中。然后把pi值加载到FPU堆栈的顶部，这使半径的平方值向下移动到ST(1)的位置。然后使用FMUL指令使FPU内的这两个值相乘。

从堆栈的顶部取出输出值并且赋值给C变量area。因为使用了ST(1)寄存器，但是没有把它赋值为输出值，所以必须在改动的寄存器列表中列出它，以便编译器知道在处理完成之后清空它。

### 13.3.10　处理跳转

内联汇编语言代码也可以包含定义其中位置的标签。可以实现一般的汇编条件分支和无条件分支，跳转到定义的标签。

jmptest.c程序演示这种情况：

```
/* jmptest.c - An example of using jumps in inline assembly */
```

```
#include <stdio.h>

int main()
{
   int a = 10;
   int b = 20;
   int result;

   asm("cmp %1, %2\n\t"
       "jge greater\n\t"
       "movl %1, %0\n\t"
       "jmp end\n"
       "greater:\n\t"
       "movl %2, %0\n"
       "end:"
       :"=r"(result)
       :"r"(a), "r"(b));

   printf("The larger value is %d\n", result);
   return 0;
}
```

内联汇编代码在指令中定义两个标签，CMP指令用于比较加载到寄存器中的两个输入值，然后使用JGE指令。JMP指令用于无条件跳转到内联汇编代码的结尾处。

编译器生成的汇编代码包含标签和指令：

```
        movl    -4(%ebp), %edx
        movl    -8(%ebp), %eax
#APP
        cmp %edx, %eax
        jge greater
        movl %edx, %eax
        jmp end
greater:
        movl %eax, %eax
end:
#NO_APP
        movl    %eax, -12(%ebp)
```

在内联汇编代码中使用标签时有两个限制。第一个限制是只能跳转到相同的asm段内的标签。不能从一个asm段跳转到另一个asm段中的标签。

第二个限制更加复杂一些。jmptest.c程序使用标签greater和end。但是，这样有个潜在的问题。查看汇编后的代码清单，可以发现内联汇编标签也被编码到了最终汇编后的代码中。这意味着如果在C代码中还有另一个asm段，就不能再次使用相同的标签，否则会因为标签的重复使用而导致错误消息。还有，如果试图整合使用C关键字（比如函数名称或者全局变量）的标签，也会导致错误。

这个问题有两个解决方案。最简单的解决方案是在不同的asm段中使用不同的标签。如果你手工编码每个asm段，这是可行的方案。

如果使用相同的asm段（比如，如果就像后面13.4节中讲解的那样声明宏），就不能改变内联汇编代码中的标签。这时的解决方案是使用局部标签。

条件分支和无条件分支都允许指定一个数字加上方向标志作为标签，方向标志指出处理器应该向哪个方向查找数字型标签。第一个遇到的标签会被采用。可以使用jmptest2.c程序演示这种情况：

```
/* jmptest2.c - An example of using generic jumps in inline assembly */
#include <stdio.h>

int main()
{
   int a = 10;
   int b = 20;
   int result;

   asm("cmp %1, %2\n\t"
       "jge 0f\n\t"
       "movl %1, %0\n\t"
       "jmp 1f\n "
       "0:\n\t"
       "movl %2, %0\n "
       "1:"
       :"=r"(result)
       :"r"(a), "r"(b));

   printf("The larger value is %d\n", result);
   return 0;
}
```

标签被替换为0: 和1:。JGE和JMP指令使用修饰符f指出从跳转指令向前查找标签。要想向后移动，必须使用修饰符b。

## 13.4 使用内联汇编代码

虽然可以在C程序中的任何位置放置内联汇编代码，但是大多数程序员把内联汇编代码用作宏函数。C宏函数可以声明包含一个函数的单一宏。在主程序中引用宏时，宏被扩展为宏定义的完整函数。本节介绍如何在C程序中创建内联汇编宏。

### 13.4.1 什么是宏

在C和C++程序中，宏被用于定义从常量值到复杂函数的任何内容。使用#define语句定义宏。#define语句的格式如下：

```
#define NAME expression
```

按照约定，总是使用大写字母定义宏名称NAME（这样确保不会和C库函数冲突）。值expression可以是常量的数字型或者字符串值。

如果读者曾经做过很多C或者C++编码工作，很可能早已熟悉如何定义常量宏了。常量宏把特定值赋值给宏名称。然后可以在整个程序中使用宏名称代表这个值。

可以把宏定义为数字型的值，如下所示：

```
#define MAX_VALUE 1024
```

在程序代码中使用MAX_VALUE时，编译器把它替换为和它相关联的值：

```
data = MAX_VALUE;
if (size > MAX_VALUE)
```

宏的值不被当作变量对待，因为不能改变它。在整个程序中它保持为常量值。但是，可以在数字型的方程式中使用它：

```
data = MAX_VALUE / 4;
```

C宏的另一个方面是宏函数。这在下一节中介绍。

### 13.4.2 C宏函数

定义值时使用常量宏很方便，而在整个程序中可以利用宏函数节省键入代码的时间。在程序的开头，可以把整个函数赋值给宏，并且可以在程序中的任何位置使用它。

宏函数定义输入值和输出值，然后定义处理输入值并且生成输出值的函数。宏函数的格式如下：

```
#define NAME(input values, output value) (function)
```

输入值是用逗号分隔的用作函数输入的变量的列表。函数定义如何处理输入值来生成输出值。

宏被定义为单一的文本行。宏函数可能创建非常长的代码行。为了帮助使宏具有更高的可读性，可以使用行继续字符（反斜线）分割函数。下面是一个简单的C宏函数的例子：

```
#define SUM(a, b, result) \
        ((result) = (a) + (b))
```

宏SUM被定义为要求两个输入值，并且生成单一输出值，输出值是两个输入值相加的结果。在程序中使用宏函数SUM()时，编译器会把它扩展为完整的宏函数定义。

注意这种方式与标准的C函数相反，C函数用于节省编码空间，这一点很重要。在对代码进行汇编之前，编译器扩展完整的宏函数，这会创建更大型的代码。

mactest1.c程序中显示C宏函数的例子：

```
/* mactest1.c - An example of a C macro function */
#include <stdio.h>

#define SUM(a, b, result) \
            ((result) = (a) + (b))

int main()
{
   int data1 = 5, data2 = 10;
   int result;
   float fdata1 = 5.0, fdata2 = 10.0;
   float fresult;

   SUM(data1, data2, result);
   printf("The result is %d\n", result);
   SUM(1, 1, result);
   printf("The result is %d\n", result);
   SUM(fdata1, fdata2, fresult);
   printf("The floating result is %f\n", fresult);
   SUM(fdata1, fdata2, result);
```

```
    printf("The mixed result is %d\n", result);
    return 0;
}
```

mactest1.c范例程序中有几个有趣的地方。首先注意，宏函数中定义的变量完全独立于程序中定义的结果变量。可以在宏函数SUM()中使用任何变量。

其次注意，相同的宏函数SUM()可以处理整数输入值、数字型输入值、浮点输入值，甚至还可以处理混合的输入值和输出值！可以发现宏函数的通用性有多么强。现在是把它应用于内联汇编函数的时候了。

如果希望看到带有扩展的宏代码行的代码，可以在编译时使用-E命令行选项。

### 13.4.3 创建内联汇编宏函数

和对待C宏函数一样，可以声明包含内联汇编代码的宏函数。内联汇编代码必须使用扩展asm格式，以便可以定义正确的输入值和输出值。因为程序中可以多次使用宏函数，所以还应该在汇编代码需要的任何分支语句中使用数字型的标签。

下面是定义内联汇编宏函数的一个例子：

```
#define GREATER(a, b, result) ({ \
    asm("cmp %1, %2\n\t" \
        "jge 0f\n\t" \
        "movl %1, %0\n\t" \
        "jmp 1f\n " \
        "0:\n\t" \
        "movl %2, %0\n " \
        "1:" \
        :"=r"(result) \
        :"r"(a), "r"(b)); })
```

输入变量a和b被赋值给寄存器，以便可以在CMP指令中使用它们。JGE和JMP指令使用数字型的标签，以便可以在程序中多次使用宏函数而不会出现重复的汇编标签。从包含两个输入值中较大值的寄存器复制结果变量。注意asm语句必须括在一对花括号中，以便指出语句的开头和结尾。如果没有花括号，每次在C代码中使用宏时，编译器都会生成错误消息。

mactest2.c程序演示在C程序中使用这个宏：

```
/* mactest2.c - An example of using inline assembly macros in a program */
#include <stdio.h>

#define GREATER(a, b, result) ({ \
    asm("cmp %1, %2\n\t" \
        "jge 0f\n\t" \
        "movl %1, %0\n\t" \
        "jmp 1f\n\t" \
        "0:\n\t" \
        "movl %2, %0\n\t" \
        "1:" \
        :"=r"(result) \
        :"r"(a), "r"(b)); })

int main()
{
```

```
    int data1 = 10;
    int data2 = 20;
    int result;

    GREATER(data1, data2, result);
    printf("a = %d, b = %d    result: %d\n", data1, data2, result);
    data1 = 30;
    GREATER(data1, data2, result);
    printf("a = %d, b = %d    result: %d\n", data1, data2, result);
    return 0;
}
```

## 13.5 小结

本章讨论如何在C和C++程序之内使用汇编语言代码。内联汇编代码技术可以把汇编语言函数放置在C或者C++程序中，把程序变量传递给汇编语言代码，然后把汇编语言代码生成的输出存放到C程序的变量中。

C语言的asm语句包含汇编语言代码，这些代码被放进从C程序代码编译出的汇编语言程序中。asm语句有两种格式。基本asm格式可以直接编写汇编语言指令的代码，使用C全局变量作为输入值和输出值。

扩展asm格式提供高级技术，用于把输入值传递给汇编代码，以及把输出值传送给C程序代码。使用扩展asm格式，可以把任何类型的C数据（比如局部变量）传递给寄存器或者内存位置。输入值可以赋值给特定的寄存器，或者可以允许编译器按照需要把输入值赋值给寄存器。类似地，输出值可以赋值给寄存器或者内存位置。可以使用众多特性控制如何在内联汇编语言代码中使用变量。

经常使用C宏函数定义asm段中的内联汇编语言代码。C宏函数使用一种格式定义函数名称、用到的输入值和用到的输出值以及asm段函数。每次在主程序中调用宏函数时，编译器会扩展内联汇编语言代码。

下一章更加深入地讨论在混合的程序设计环境中使用汇编语言。除了内联汇编语言代码之外，还可以创建C和C++程序可以利用的完整汇编语言库。下一章讨论并且演示这种技术。

# 第14章　调用汇编库

前一章演示如何通过内联汇编程序设计的方法，把汇编语言代码整合到C程序中。本章讨论把汇编语言代码整合到C或者C++程序中的另一种途径。

C和C++程序都可以直接调用汇编语言函数，把输入值传递给函数，并且从函数获得输出值。本章讨论如何完成这样的操作。首先，回顾C样式的汇编语言函数的基础知识。然后，学习如何编译C程序和汇编语言函数。下一节讲解如何在C程序和汇编语言函数之间传递值，包括输入值和输出值。然后学习如何把汇编语言函数组合成可以和C程序一起使用的通用库。本章讨论静态库和动态库，并且分别提供每种库的范例。最后，讨论在C程序中调试汇编语言函数的主题。

## 14.1　创建汇编函数

第11章“使用函数”演示了如何创建可以在任何汇编语言程序中使用的汇编语言函数。相同的技术可以为高级的C和C++程序提供汇编语言函数。本节简要回顾如何创建汇编语言函数。

如果回顾第11章，会发现有很多把输入值传递给汇编函数的方法，以及很多获得输出结果的方法。C程序使用特定格式把输入值传递到程序堆栈中，并且从EAX寄存器获得结果。

如果希望汇编语言函数和C以及C++程序一起工作，就必须显式地遵守C样式的函数格式。这就是说所有输入变量都必须从堆栈中读取，并且大多数输出值都返回到EAX寄存器中（本章后面将更加详细地讨论例外的情况）。

图14-1显示把输入值存放到堆栈中的方式，以及汇编语言函数如何访问它们。

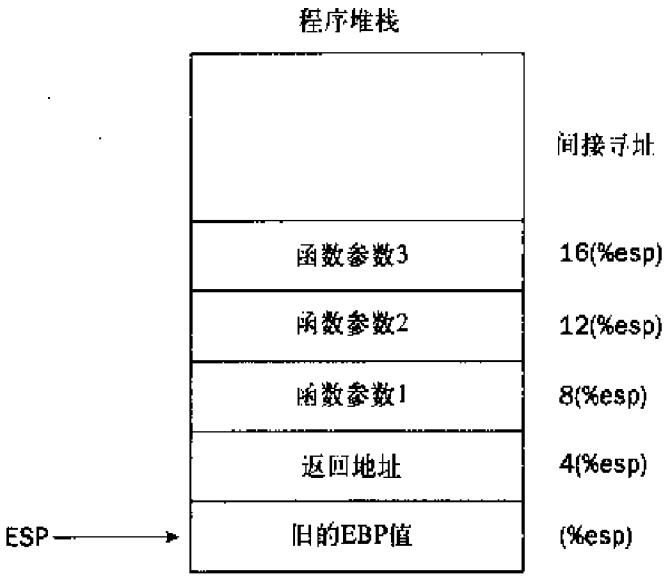



图 14-1

EBP寄存器用作访问堆栈中的值的基址指针。调用汇编语言函数的程序必须知道输入值按照什么顺序存放在堆栈中，以及每个输入值的长度（和数据类型）。

在汇编函数代码中，C样式函数对于可以修改哪些寄存器和函数必须保留哪些寄存器有着特定的规则。如果必须保留的寄存器在函数中被修改了，那么必须恢复寄存器的原始值，否则在执行返回发出调用的C程序时也许会出现不可预料的结果。在函数中可以安全地使用MMX和SSE寄存器，但是使用通用寄存器和FPU寄存器时必须谨慎。下表列出寄存器在函数中的状态。

如下表所示，被调用的函数必须保留EBX、EDI、ESI、EBP和ESP寄存器。这就要求在执行函数代码之前把寄存器的值压入堆栈，并且在函数准备返回调用程序时把它们弹出堆栈。这通

常是按照标准的开头和结尾格式完成的（和第11章中用于汇编语言函数的格式类似）。

| 寄存器 | 状 态 |
|---|---|
| EAX | 用于保存输出值，但是可能在函数返回之前被修改 |
| EBX | 用于指向全局偏移表；必须保留 |
| ECX | 在函数中可用 |
| EDX | 在函数中可用 |
| EBP | C程序使用它作为堆栈基址指针；必须保留 |
| ESP | 在函数中用于指向新的堆栈位置；必须保留 |
| EDI | C程序使用它作为局部寄存器；必须保留 |
| ESI | C程序使用它作为局部寄存器；必须保留 |
| ST(0) | 用于保存浮点输出值，但是可能在函数返回之前被修改 |
| ST(1) - ST(7) | 在函数中可用 |

C函数调用的汇编语言函数的基本模板如下：

```
.section .text
.type func, @function
func:
   pushl %ebp
   movl %esp, %ebp
   subl $12, %esp
   pushl %edi
   pushl %esi
   pushl %ebx

    <function code>

   popl %ebx
   popl %esi
   popl %edi
   movl %ebp, %esp
   popl %ebp
   ret
```

这个代码模板可以用于C或者C++函数使用的所有汇编语言函数。当然，如果特定的函数不改变EBX、ESI或者EDI寄存器，可以省略相关的PUSH和POP指令。

注意开头中包含的SUBL指令。如果回顾第11章，可以发现它用于为函数中的局部变量保留堆栈空间。这条指令在堆栈中保留12个字节的内存空间。这段空间可以用于保存3个4字节的数据值。如果局部变量需要更多空间，就必须从ESP的值减去空间的长度。在函数中，相对于EBP寄存器引用局部变量。例如，如果第一个局部变量为4字节值，那么它的位置就是-4（%ebp）。用-8（%ebp）引用第二个变量，用-12（%ebp）引用第三个变量。

但是，C程序调用的汇编语言函数也许还声明它们自己的.data和.bss段用于存储数据。在编译时，这些内存区域将和C程序的内存需求结合在一起。指针可以传递回调用程序以便访问存储在这些内存位置中的任何数据（在后面的14.3节中演示）。

## 14.2 编译C和汇编程序

当主应用程序包含在C或者C++程序中，而且函数包含在汇编语言程序中时，必须一起编译

它们生成单一的可执行文件。GNU的C编译器提供可以用于生成最终程序文件的若干选择。本节介绍可以用于从源代码文件手动地创建可执行文件的两种方法。

### 14.2.1 编译汇编源代码文件

编译包含汇编语言函数的C程序时，编译器必须知道如何访问函数。如果编译器不能解析程序中使用的函数，就会产生错误消息：

```
$ gcc -o mainprog mainprog.c
/tmp/cc9hGAnP.o: In function `main':
/tmp/cc9hGAnP.o(.text+0xc): undefined reference to `asmfunc'
collect2: ld returned 1 exit status
$
```

mainprog.c程序对名为asmfunc()的汇编语言函数发出函数调用。编译这个程序时，连接器不能解析函数名称，并且生成错误消息。

为了解决这个问题，在编译时，编译器必须可以利用汇编语言函数和C程序代码。完成这一工作的一种方式是把汇编语言函数的源代码文件包含在编译器命令行中。编译器命令行应该像这样：

```
$ gcc -o mainprog mainprog.c asmfunc1.s asmfunc2.s asmfunc3.s
```

编译器的输出是单一可执行文件mainprog，它包含主程序和所有汇编语言函数所需的所有程序代码。这种方法不会生成中间目标文件，只有最终的可执行文件。取决于如何处理汇编语言函数，这或许是，或许不是好办法。下一节介绍如何使用单独的汇编语言目标文件编译程序。

### 14.2.2 使用汇编目标代码文件

不在编译时对汇编语言函数进行汇编，可以单独地汇编它们，并且在C程序编译器命令中引用目标文件。GNU编译器能够从目标代码文件获得汇编语言函数，并且把它们连接到主程序以便生成可执行文件。

创建汇编语言函数的目标文件时，不必使用ld命令连接代码，因为它本身不做什么操作，但是必须使用as命令进行汇编。

如果试图孤立地连接汇编语言函数，连接器就会产生错误，指出没有定义_start标签，但是会假设.text段的开头是程序的开头。

创建了目标文件之后，可以把它和C主程序一起包含在编译器的命令行中：

```
$ as -o asmfunc.o asmfunc.s
$ gcc -o mainprog mainprog.c asmfunc.o
```

这生成可执行文件mainprog，它包含C主程序和汇编语言函数的代码。如果C程序中使用的汇编语言函数有多个，可以单独地汇编所有函数，然后把它们添加到编译器的命令行中：

```
$ gcc -o mainprog mainprog.c asmfunc1.o asmfunc2.o asmfunc3.o
```

如果改动了一个汇编语言函数，就必须汇编它并且重新编译主程序。虽然这对于小型程序似乎是简单的工作，但是对于使用位于单独文件中的数十个函数的大型应用程序，这个过程就

变得繁琐了。可以使用的非常方便的工具是GNU的make实用程序。

GNU的make实用程序使得可以创建定义文件（称为makefile），这种文件定义使用什么代码文件创建可执行文件。make程序运行时，它自动汇编和编译创建可执行文件所需的程序文件。如果make运行之后改动了任何程序文件，只要汇编（或者编译）这个文件从而创建新的可执行文件即可。

### 14.2.3 可执行文件

编译过程的最终产品是可以运行在Linux系统上的可执行文件。可执行文件包含主程序的指令，还有主程序调用的所有函数的指令。可以使用objdump程序查看可执行文件的各个部分，以便查看主程序汇编语言代码如何和汇编语言函数代码进行交互。

为了演示这种情况，可以一起创建和编译汇编语言函数和C语言主程序的例子。asmfunc.s程序创建一个简单的汇编语言函数的例子，它在控制台屏幕显示一条消息，以便证实它的运行：

```
# asmfunc.s - An example of a simple assembly language function
.section .data
testdata:
   .ascii "This is a test message from the asm function\n"
datasize:
   .int 45
.section .text
.type asmfunc, @function
.globl asmfunc
asmfunc:
   pushl %ebp
   movl %esp, %ebp
   pushl %ebx

   movl $4, %eax
   movl $1, %ebx
   movl $testdata, %ecx
   movl datasize, %edx
   int $0x80

   popl %ebx
   movl %ebp, %esp
   popl %ebp
   ret
```

asmfunc.s程序使用标准C样式函数的开头和结尾代码。因为没有修改EDI和ESI寄存器，所以没有把它们压入堆栈，并且没有声明局部变量。因为函数代码中修改了EBX寄存器，所以它被压入堆栈并且在函数的结尾恢复。

这个函数把输出文本消息存储在.data段中，并且把消息长度声明为整数值。使用Linux系统调用write()（系统调用值为4），使用文件描述符1在STDOUT显示消息。

范例C程序mainprog.c中使用这个函数：

```
/* mainprog.c - An example of calling an assembly function */
#include <stdio.h>

int main()
```

```
{
        printf("This is a test.\n");
        asmfunc();
        printf("Now for the second time.\n");
        asmfunc();
        printf("This completes the test.\n");
        return 0;
}
```

mainprog.c程序通过名称调用汇编语言函数，使用括号表明这个名称是函数。因为这个函数不需要任何输入值，所以没有在函数中提供。类似地，这个函数不使用返回值，所以C程序中没有声明。

使用下面的命令创建并运行可执行程序:

```
$ gcc -o mainprog mainprog.c asmfunc.s
$ ./mainprog
This is a test.
This is a test message from the asm function
Now for the second time.
This is a test message from the asm function
This completes the test.
$
```

程序像预期那样工作。每次调用汇编语言函数时，它生成消息，然后把控制返回给主程序。

既然已经具有了一个完整的可执行程序，可以使用objdump程序查看编译后的代码:

```
$ objdump -D mainprog > dump
```

文件dump包含可执行文件完整的反汇编后的源代码。可以使用系统上的任何文本编辑器查看它。可以注意到源代码中有若干段。这些段中的一个称为main。这个段包含用来实现C程序代码的Mandrake Linux系统上生成的汇编语言代码（你的Linux系统编译器可能生成不同的代码）:

```
08048460 <main>:
 8048460:       55                      push   %ebp
 8048461:       89 e5                   mov    %esp,%ebp
 8048463:       83 ec 08                sub    $0x8,%esp
 8048466:       83 ec 0c                sub    $0xc,%esp
 8048469:       68 34 85 04 08          push   $0x8048534
 804846e:       e8 c9 fe ff ff          call   804833c <_init+0x58>
 8048473:       83 c4 10                add    $0x10,%esp
 8048476:       e8 31 00 00 00          call   80484ac <asmfunc>
 804847b:       83 ec 0c                sub    $0xc,%esp
 804847e:       68 45 85 04 08          push   $0x8048545
 8048483:       e8 b4 fe ff ff          call   804833c <_init+0x58>
 8048488:       83 c4 10                add    $0x10,%esp
 804848b:       e8 1c 00 00 00          call   80484ac <asmfunc>
 8048490:       83 ec 0c                sub    $0xc,%esp
 8048493:       68 5f 85 04 08          push   $0x804855f
 8048498:       e8 9f fe ff ff          call   804833c <_init+0x58>
 804849d:       83 c4 10                add    $0x10,%esp
 80484a0:       b8 00 00 00 00          mov    $0x0,%eax
 80484a5:       89 ec                   mov    %ebp,%esp
 80484a7:       5d                      pop    %ebp
 80484a8:       c3                      ret
 80484a9:       90                      nop
```

```
 80484aa:       90                      nop
 80484ab:       90                      nop
```

objdump程序从可执行文件生成反汇编后的汇编语言代码。第一列是程序内存空间中的内存位置。第二列显示汇编语言代码生成的IA-32指令代码。最后一列显示反汇编后的汇编语言代码。

注意代码使用CALL指令调用asmfunc函数的两个位置。这个函数的汇编代码在另一个段中，称为asmfunc:

```
080484ac <asmfunc>:
 80484ac:       55                      push   %ebp
 80484ad:       89 e5                   mov    %esp,%ebp
 80484af:       53                      push   %ebx
 80484b0:       b8 04 00 00 00          mov    $0x4,%eax
 80484b5:       bb 01 00 00 00          mov    $0x1,%ebx
 80484ba:       b9 8c 95 04 08          mov    $0x804958c,%ecx
 80484bf:       8b 15 b9 95 04 08       mov    0x80495b9,%edx
 80484c5:       cd 80                   int    $0x80
 80484c7:       5b                      pop    %ebx
 80484c8:       89 ec                   mov    %ebp,%esp
 80484ca:       5d                      pop    %ebp
 80484cb:       c3                      ret
 80484cc:       90                      nop
 80484cd:       90                      nop
 80484ce:       90                      nop
 80484cf:       90                      nop
```

## 14.3 在C程序中使用汇编函数

调用汇编语言函数的C或者C++程序必须知道用于把输入值传递给汇编函数的正确格式，以及如何处理任何返回值。本节演示如何从C和C++程序访问汇编语言函数的数据。

### 14.3.1 使用整数返回值

最基本的汇编语言函数调用把32位整数值返回到EAX寄存器中。调用函数获得这个值，它必须把返回值作为整数赋值给C变量:

```
int result = function();
```

C程序生成的汇编语言代码提取存放在EAX寄存器中的值，并且把它传送到分配给C变量名称的内存位置（通常是堆栈中的局部变量）。这个C变量包含来自汇编语言函数的返回值，可以在整个C程序中像以往那样使用它。

下面是汇编语言函数和使用它的C程序的例子。首先，汇编语言程序square.s定义一个函数，它需要一个整数输入值，求它的平方，然后把结果返回到EAX寄存器中:

```
# square.s - An example of a function that returns an integer value
.type square, @function
.globl square
square:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  imull %eax, %eax
```

```
movl %ebp, %esp
popl %ebp
ret
```

从堆栈读取输入值并且将它存放到EAX寄存器中。IMUL指令用于使这个值和自己相乘，结果存放在EAX寄存器中。因为这个函数不影响EBX、EDI和ESI寄存器，所以在开头和结尾中没有包含它们。另外，这个函数不需要局部数据存储，所以没有修改ESP寄存器。

相应的C程序inttest.c使用汇编语言函数square:

```
/* inttest.c - An example of returning an integer value */
#include <stdio.h>

int main()
{
        int i = 2;
        int j = square(i);
        printf("The square of %d is %d\n", i, j);

        j = square(10);
        printf("The square of 10 is %d\n", j);
        return 0;
}
```

C程序inttest.c使用两个不同的输入值调用求平方的函数。第一次它使用C变量i，这个变量的值为2。第二次它使用立即数值10。在两个实例中，输入值都作为整数值被传递给汇编语言函数并且被处理。返回值被赋值给C变量j。可以和任何其他的C变量一样处理变量j。

可以独立于C程序对汇编语言函数进行汇编，也可以在gcc编译器命令行中一起编译:

```
$ gcc -o inttest inttest.c square.s
$ ./inttest
The square of 2 is 4
The square of 10 is 100
$
```

对于64位的长整数值，返回值存放在EDX：EAX寄存器对中。

### 14.3.2 使用字符串返回值

处理返回字符串值的函数要困难一些。和返回整数值到EAX寄存器中的函数不同，这种函数不能把整个数据字符串返回到EAX寄存器中（当然，除非字符串的长度是4个字符）。

替换的做法是，返回字符串的函数返回指向字符串存储位置的指针。调用这个函数的C或者C++程序必须使用指针变量保存返回值。然后可以通过指针值访问字符串，如图14-2所示。

字符串值被包含在函数的内存空间之内，但是主程序可以访问它，因为函数内存空间包含在主程序的内存空间之内。函数返回的32位指针值是字符串开始位置所在的内存地址。

在C和C++程序中处理字符串值时，记住字符串必须使用空字符结尾。

C和C++语言在变量名称前面使用星号表明这个变量包含一个指针。可以创建指向任何数据类型的指针，但是对于字符串，想要创建指向数据类型char的指针。变量的声明应该像这样:

```
char *result;
```

这创建称为result的变量，可以使用它包含指向字符串的指针。

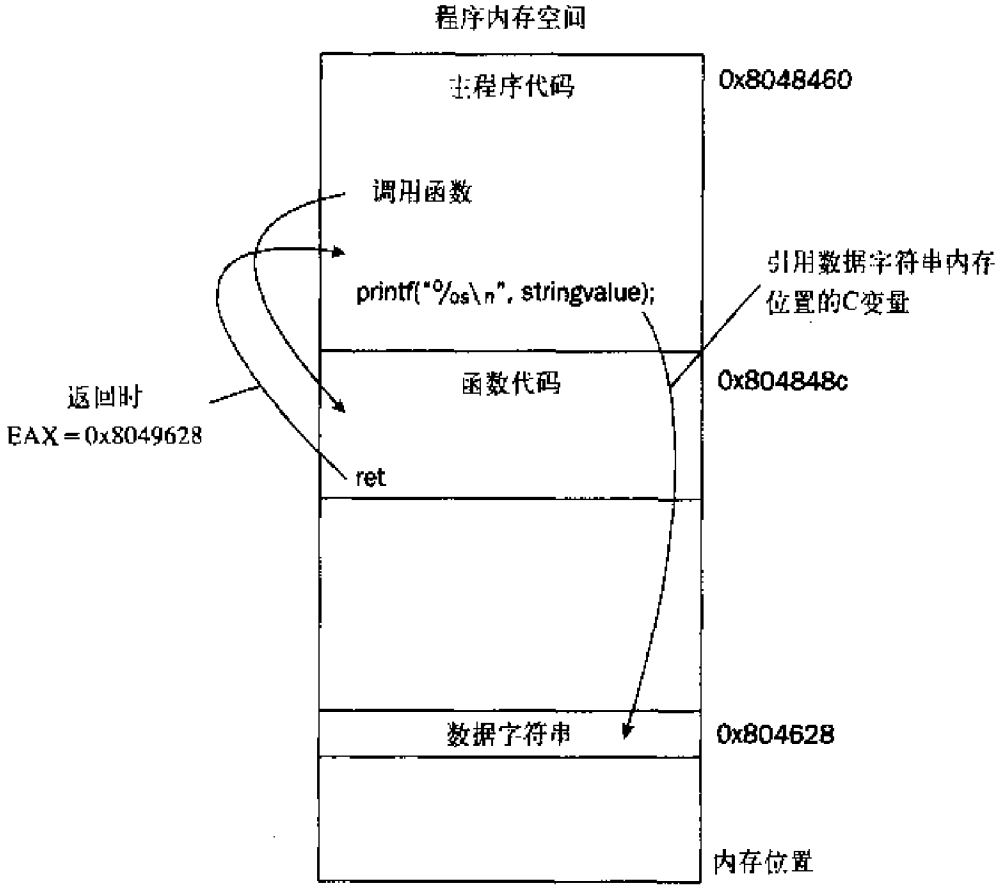



图 14-2

不幸的是，默认情况下，C程序假设函数的返回值是整数值。必须通知编译器这个函数将返回字符串的指针。创建函数调用的原型（prototype）将完成这个任务。

原型在使用函数之前定义函数格式，以便C编译器知道如何处理函数调用。原型定义函数需要的输入值的数据类型，还有返回值的数据类型。不定义特定的值，只定义需要的数据类型。原型的例子如下：

```
char *function1(int, int);
```

这个原型定义名为function1的函数，它需要两个输入值，它们的数据类型都是integer。返回数据类型被定义为指向数据类型char的指针。要记住函数模板后面有分号，这很重要。没有这个分号，编译器就会假设你在代码中定义的是整个函数，并且会生成错误消息。如果函数没有使用任何输入值，仍然必须指定数据类型void:

```
char *function(void);
```

在源代码中，原型必须出现在main()段之前，并且通常放置在所有#include和#define语句之后。

作为使用字符串返回值的例子，下面是一个汇编语言函数。它使用CPUID指令，并且返回一个指针，指向从处理器获得的包含CPUID字符串的字符串。程序cpuidfunc.s执行汇编语言函数部分。

```
# cpuidfunc.s - An example of returning a string value
.section .bss
   .comm output, 13
.section .text
.type cpuidfunc, @function
.globl cpuidfunc
cpuidfunc:
   pushl %ebp
   movl %esp, %ebp
   pushl %ebx
   movl $0, %eax
   cpuid
   movl $output, %edi
   movl %ebx, (%edi)
   movl %edx, 4(%edi)
   movl %ecx, 8(%edi)
   movl $output, %eax
   popl %ebx
   movl %ebp, %esp
   popl %ebp
   ret
```

这个汇编语言程序为输出字符串定义一个缓冲区区域，这很重要。这个值将包含在最终程序中，并且将被主C程序访问。在这个例子中，在.bss内存段中声明值output。它被声明为带有一个附加字节。这个字节将包含结尾的空字符（一个0），以使这个值是正确的C字符串值。使用0初始化.bss内存区域，以便我们能够保证它将包含0值。

因为CPUID指令修改EBX寄存器，所以在执行函数代码之前，函数代码把EBX寄存器的原始值压入堆栈，并且在函数退出之前把它弹出堆栈。

stringtest.c程序演示使用cpuid函数的C程序：

```
/* stringtest.c - An example of returning a string value */
#include <stdio.h>

char *cpuidfunc(void);

int main()
{
        char *spValue;
        spValue = cpuidfunc();
        printf("The CPUID is: '%s'\n", spValue);
        return 0;
}
```

源代码中，在主程序代码之前，首先定义函数原型。在主程序中使用函数时，返回值被赋值给字符指针变量spValue。然后可以在任何类型的C字符串处理函数中使用这个变量。

和整数函数类似，可以单独地编译cpuidfunc.s程序，然后在编译器的命令行中使用目标文件；也可以在编译器的命令行中包含汇编语言的源代码文件。下面是单独对函数进行汇编的例子：

```
$ as -o cpuidfunc.o cpuidfunc.s
$ gcc -o stringtest stringtest.c cpuidfunc.o
$ ./stringtest
The CPUID is: 'GenuineIntel'
$
```

字符串值不是具有返回的指针值的唯一情况。函数可以使用相同的技术返回指向整数和浮点值的指针。

### 14.3.3 使用浮点返回值

浮点返回值是特殊的情况。整数和字符串返回值都使用EAX寄存器把值从汇编语言函数返回到发出调用的C程序。如果希望返回浮点值，情况稍微有些区别。

C样式的函数不使用EAX寄存器，而是使用ST(0)寄存器在函数之间交换浮点值。函数把返回值存放到FPU堆栈中，然后调用程序负责把返回值弹出堆栈并且把它赋值给变量。

因为把浮点值存放到FPU堆栈中时，总会把浮点值转换为扩展双精度浮点值，所以原始值是什么浮点数据类型，以及C程序使用什么浮点数据类型包含结果值不重要。由FPU完成适当的转换。

C和C++程序使用两种数据类型表示浮点值：

- float：表示单精度浮点值
- double：表示双精度浮点值

以上每一种类型都可以用于从FPU堆栈获得结果值。汇编语言函数使用什么精度把值存放到FPU堆栈中不重要。

在C程序中使用返回浮点值的函数时，必须定义函数原型。和字符串返回值的原型相同的原则在这里也适用：

```
float function1(float, float, int);
```

这个例子定义的函数需要3个输入值（两个单精度浮点值和一个整数值），并且返回一个单精度浮点值。如果C变量必须使用双精度浮点值，就必须使用返回类型double：

```
double function1(double, int);
```

作为使用浮点返回值的例子，函数areafunc.s使用C返回值样式编写：

```
# areafunc.s - An example of a floating point return value
.section .text
.type areafunc, @function
.globl areafunc
areafunc:
   pushl %ebp
   movl %esp, %ebp

   fldpi
   filds 8(%ebp)
   fmul %st(0), %st(0)
   fmul %st(1), %st(0)
   movl %ebp, %esp
   popl %ebp
   ret
```

函数areafunc.s把pi值加载到FPU堆栈中，然后从堆栈把整数输入值（半径值）加载到FPU堆栈中。第一个值（整数半径值）和自己相乘，然后和pi值相乘，结果存放在FPU堆栈寄存器ST(0)中。同样，因为这个函数中不使用EBX、EDI和ESI寄存器，所以不使用PUSH和POP指令专门保留它们的值。

在函数的结尾，要确保返回值在ST(0)寄存器中，这很重要。可能必须编写函数代码专门实现这一点，或者只需在运算完成时把浮点值压入FPU堆栈的顶部。

floattest.c程序是使用函数areafunc的范例C程序：

```
/* floattest.c - An example of using floating point return values */
#include <stdio.h>

float areafunc(int);

int main()
{
        int radius = 10;
        float result;
        result = areafunc(radius);
        printf("The result is %f\n", result);

        result = areafunc(2);
        printf("The result is %f\n", result);
        return 0;
}
```

像通常一样，必须确保编译floattest.c程序时areafunc的代码是可用的：

```
$ gcc -o floattest floattest.c areafunc.o
$ ./floattest
The result is 314.159271
The result is 12.566371
$
```

### 14.3.4 使用多个输入值

处理汇编函数时，需要传递的输入值也许会超过一个。传递给汇编语言函数的输入值的数量是没有限制的。在调用函数之前，每个输入值都要存放在堆栈中。

使用多个输入值时，必须谨慎地注意按照什么顺序把它们传递给函数。输入值按照C函数中列出的从左到右的顺序存放。因此，函数

```
int i = 10;
int j = 20;
result = function1(i, j);
```

将按照图14-3中显示的顺序把输入值存放到堆栈中。

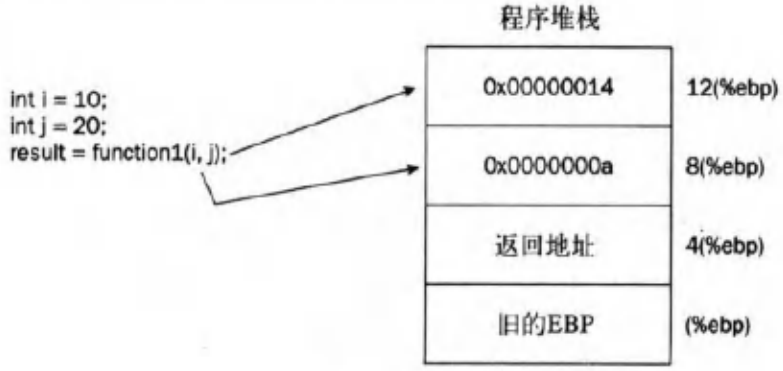



图 14-3

第二个输入值j先存放在堆栈中，接下来存放第一个输入值i。记住堆栈是向下增长的。在函数中使用位置8（%ebp）引用第一个输入值，使用位置12（%ebp）引用第二个输入值。

可以在一个简单的汇编语言函数例子中演示这种情况。greater.s函数接受两个整数输入值，并且返回其中的最大值：

```
# greater.s - An example of using multiple input values
.section .text
.globl greater
greater:
   pushl %ebp
   movl %esp, %ebp
   movl 8(%ebp), %eax
   movl 12(%ebp), %ecx
   cmpl %ecx, %eax
   jge end
   movl %ecx, %eax
end:
   movl %ebp, %esp
   popl %ebp
   ret
```

greater.s程序从两个输入值在堆栈中的位置获得它们，把它们加载到寄存器中。使用CMP指令比较两个寄存器。使用JGE指令确定EAX寄存器（包含第一个输入值）是否大于或者等于ECX寄存器（包含第二个输入值）。如果ECX寄存器的值更大，就把它复制到EAX寄存器中，然后函数返回调用程序。

multtest.c程序演示使用greater函数的C程序：

```
/* multtest.c - An example of using multiple input values */
#include <stdio.h>

int main()
{
        int i = 10;
        int j = 20;
        int k = greater(i, j);
        printf("The larger value is %d\n", k);
        return 0;
}
```

multtest.c程序把两个整数值存放在堆栈中，并且把返回值赋值给变量k。

### 14.3.5 使用混合数据类型的输入值

当汇编语言函数需要多个不同数据类型的输入值时，情况就变得更加复杂了。使用混合类型的输入值时会遇到两个问题：

- 调用函数可能按照错误的顺序把值存放到堆栈中。
- 汇编函数可能按照错误的顺序从堆栈读取值。

必须谨慎地处理，以便确保调用程序正确地把输入值存放到函数中，并且汇编语言函数按照正确的顺序从堆栈读取输入值。本节提供完成这些操作的几个例子。

1. 按照正确的顺序存放输入值

如果在处理使用不同数据长度的数据类型，就必须确保调用程序按照汇编语言函数读取输入值的顺序把输入值存放到堆栈中。如果某些值是4字节整数值，而其他值是8字节双精度浮点值，那么如果函数没有正确地读取它们，就会发生错误。图14-4显示当double和integer数据类型的输入值被存放到堆栈中的顺序和函数期望的顺序不同时发生的情况。

为了演示这种情况，我们创建需要一个双精度浮点输入值和一个整数输入值的汇编语言函数。我们使用testfunc.s程序：

```
# testfunc.s - An example of reading input values wrong
.section .text
.type testfunc, @function
.globl testfunc
testfunc:
   pushl %ebp
   movl %esp, %ebp

   fldl 8(%ebp)
   fimul 16(%ebp)

   movl %ebp, %esp
   popl %ebp
   ret
```

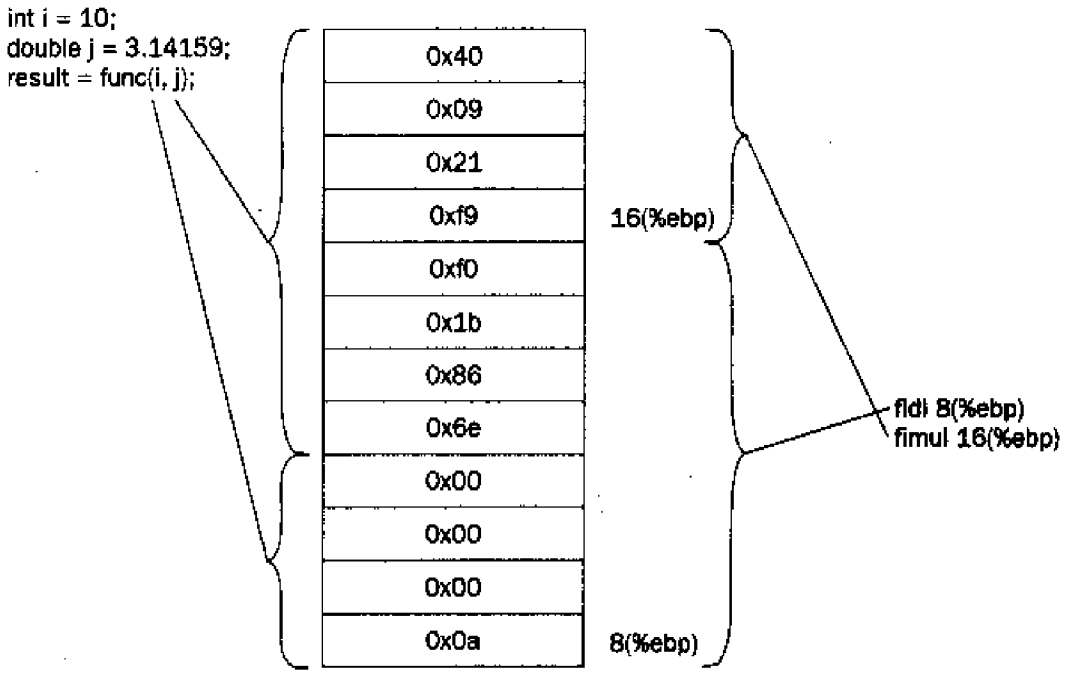



图 14-4

testfunc.s函数假设第一个输入值是8字节的双精度浮点值，并且把它加载到FPU堆栈中。然后它假设下一个输入值位于距离第一个值8字节的位置，并且是4字节整数值。这个值和FPU堆栈中的第一个值相乘。结果返回到FPU堆栈的寄存器ST(0)中。

现在我们来编写按照错误的顺序传递输入值的程序。这个程序是badprog.c：

```
/* badprog.c - An example of passing input values in the wrong order */
```

```
#include <stdio.h>

double testfunc(int, double);

int main()
{
        int data1 = 10;
        double data2 = 3.14159;
        double result;

        result = testfunc(data1, data2);
        printf("The bad result is %g\n", result);
        return 0;
}
```

这个函数原型错误地把输入值的顺序显示为首先是整数值，然后是双精度浮点值。数据被按照错误的顺序加载，结果不是我们所期望的：

```
$ ./badprog
The bad result is -
92912748224803586789953691453287600992410969054337565541302096881878852915543572895
44137729627479662146411650817673003969823230528948599424004314292795943593962339982
21138002117106443888524575385282987136508460525990489946605875479183 36.000000
$
```

哎呀，这确实错了。为了显示应该如何工作，下面是goodprog.c程序，它正确地使用testfunc函数：

```
/* goodprog.c - An example of passing input values in the proper order */
#include <stdio.h>

double testfunc(double, int);

int main()
{
        double data1 = 3.14159;
        int data2 = 10;
        double result;

        result = testfunc(data1, data2);
        printf("The proper result is %f\n", result);
        return 0;
}
```

这次函数原型正确地定义testfunc函数，并且在程序代码中正确地使用函数。程序运行时，获得的结果是正确的：

```
$ gcc -o goodprog goodprog.c testfunc.s
$ ./goodprog
The proper result is 31.415900
$
```

2. 按照正确的顺序读取输入值

除了确保调用程序按照正确的顺序存放输入值之外，还必须确保汇编函数正确地读取输入值。fpmathfunc.s程序演示从堆栈读取多个输入值。这个程序复制第9章中介绍过的fpmath.s程序，

它计算数学表达式((43.65/22) + (76.34*3.1))/((12.43*6)−(140.2/94.21)):

```
# fpmathfunc.s - An example of reading multiple input values
.section .text
.type fpmathfunc, @function
.globl fpmathfunc
fpmathfunc:
   pushl %ebp
   movl %esp, %ebp
   flds 8(%ebp)
   fidiv 12(%ebp)
   flds 16(%ebp)
   flds 20(%ebp)
   fmul %st(1), %st(0)
   fadd %st(2), %st(0)
   flds 24(%ebp)
   fimul 28(%ebp)
   flds 32(%ebp)
   flds 36(%ebp)
   fdivrp
   fsubr %st(1), %st(0)
   fdivr %st(2), %st(0)
   movl %ebp, %esp
   popl %ebp
   ret
```

函数在相对于EBP寄存器值的变址位置中读取输入值，EBP寄存器的值指向堆栈的开头。每个输入值都必须使用正确的变址值指向堆栈中正确的位置，否则就会读取错误的值。

为了进行测试，mathtest.c程序通过fpmathfunc.s函数使用赋值的数据值计算方程式:

```
/* mathtest.c - An example of using multiple input values */
#include <stdio.h>

float fpmathfunc(float, int, float, float, float, int, float, float);

int main()
{
        float value1 = 43.65;
        int value2 = 22;
        float value3 = 76.34;
        float value4 = 3.1;
        float value5 = 12.43;
        int value6 = 6;
        float value7 = 140.2;
        float value8 = 94.21;
        float result;
        result = fpmathfunc(value1, value2, value3, value4,
                            value5, value6, value7, value8);
        printf("The final result is %f\n", result);
        return 0;
}
```

fpmathfunc函数的函数原型声明函数的格式。为了把正确的值按照正确的顺序存放的过程中，输入值的顺序是至关重要的。函数返回值被声明为单精度浮点值。

可以从命令行编译并且运行这个程序:

```
$ gcc -o mathtest mathtest.c fpmathfunc.s
$ ./mathtest
The final result is 3.264907
$
```

结果和从第9章中的fpmath.s程序获得的结果相同。

## 14.4 在C++程序中使用汇编函数

在C++程序中使用汇编语言函数的规则几乎和在C程序中使用它们的规则相同。只有一处区别，但是这一区别是重要的。

默认情况下，C++程序假设在C++程序中使用的所有函数都使用C++样式的命名和调用约定。但是，程序中使用的汇编语言函数使用C语言的调用约定（参见14.1节）。必须通知编译器使用的哪些函数是C函数。这是通过extern语句完成的。

extern语句用于定义使用C调用约定的函数，它使用下面的格式：

```
extern "C"
{
    int square(int);
    float areafunc(int);
    char *cpuidfunc();
}
```

每个用到的汇编语言函数原型都必须放置在extern语句内。这确保编译器在访问函数时使用C调用约定，并且不会破坏函数名称。

externtest.cpp程序演示在C++程序中使用汇编语言函数：

```
/* externtest.cpp - An example of using assembly language functions with C++ */
#include <iostream.h>

extern "C" {
   int square(int);
   float areafunc(int);
   char *cpuidfunc(void);
}


int main()
{
   int radius = 10;
   int radsquare = square(radius);
   cout << "The radius squared is " << radsquare << endl;
   float result;
   result = areafunc(radius);
   cout << "The area is " << result << endl;
   cout << "The CPUID is " << cpuidfunc() << endl;
   return 0;
}
```

C++代码和C程序范例之间的唯一区别在于使用extern语句声明汇编语言函数（当然，除了使用cout显示输出而不是使用printf之外）。要编译这个程序，需要在命令行中包含所有需要的汇编语言目标文件：

```
$ g++ -o externtest externtest.cpp square.o areafunc.o cpuidfunc.o
$ ./externtest
The radius squared is 100
The area is 314.159
The CPUID is GenuineIntel
$
```

## 14.5 创建静态库

处理使用C程序和汇编语言函数的程序设计项目时，在从无数目标文件构建可执行文件的过程中，容易遇到困难。如果为每个汇编语言函数创建单独的目标文件，就很可能需要把数十个不同文件编译到C程序中。不仅C程序调用汇编函数，汇编函数本身也可以调用其他的汇编函数。

可以通过使用库（library）来简化组织大量汇编函数的目标文件的问题。本节介绍库如何工作，以及如何创建可以用于编译C程序的汇编函数库。

### 14.5.1 什么是静态库

在前面的14.2节中讲过，GNU的C编译器具有众多选项，用于控制如何编译汇编语言函数和C主程序。对于依赖于若干函数的C程序来说，每个函数的目标文件都可以包含在编译器的命令行中，这样编译器会使主程序包含它们。

GNU的C编译器可以不在命令行中独立地包含每个单独的函数目标文件，它允许把所有目标文件组合在单一存档文件中。在编译C主程序时，要做的所有工作就是包含单一的目标存档文件。在编译时，编译器可以从存档文件中挑选出所需的正确的目标文件，如图14-5所示。

存档文件可以用于编译任何使用存档文件中包含的任何函数的程序。这种存档文件被称为库文件（library file）。

库文件包含很多函数的目标文件。在库文件中，经常按照应用程序类型或者函数类型把函数分组在一起。单一应用程序项目中可以使用多个库文件。

这种类型的库文件称为静态的，因为库文件中包含的目标代码被编译器编译到了主程序中。函数的目标代码被编译到可执行代码中之后，可执行程序的运行就不需要库文件了。但是，这意味着程序的每个拷贝都在其中包含函数的代码。在14.6节中将看到，共享库可以帮助程序在程序之间共享目标文件，以便节省内存需求。

### 14.5.2 ar命令

在Linux环境中，使用ar命令创建静态库文件。ar命令创建可供编译器读取的函数目标文件的存档文件。可以使用ar命令的若干命令行选项，如下表所示。

| 选　项 | 描　述 |
|---|---|
| d | 从存档文件中删除文件 |
| m | 把文件移动到存档文件中 |
| p | 把存档文件中指定的文件输出到标准输出 |
| q | 快速地把文件追加到存档文件中 |
| r | 把文件插入（替换）到存档文件中 |
| t | 显示存档文件中文件的列表 |
| x | 从存档文件提取文件 |

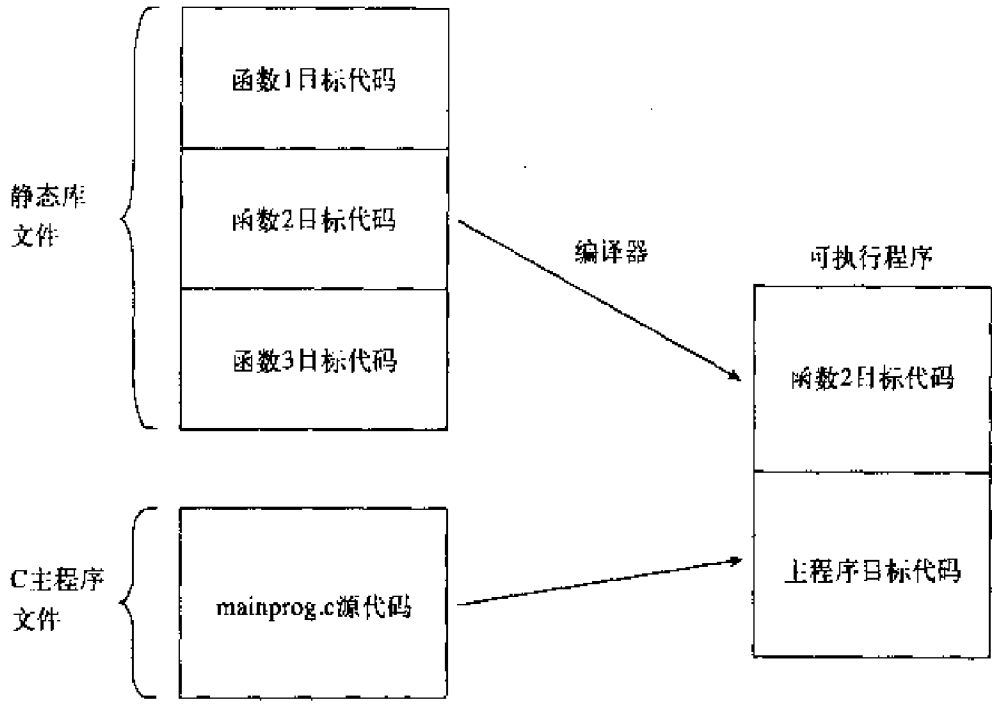



图 14-5

可以使用一个或者多个修饰符修改基本选项，如下表所示。

| 修饰符 | 描 述 |
|---|---|
| a | 把新的文件添加到存档文件中现有的文件之后 |
| b | 把新的文件添加到存档文件中现有的文件之前 |
| c | 创建新的存档文件 |
| f | 截短存档文件中的名称 |
| i | 在存档文件中现有文件之前插入新的文件 |
| P | 在存档文件中使用文件的完整路径名称 |
| s | 编写存档文件的索引 |
| u | 更新存档文件中的文件（使用新文件替换旧的） |
| v | 使用详细模式 |

ar命令的命令行格式如下:

```
ar [-]{dmpqrtx}[abcfilNoPsSuvV] [membername] [count] archive files...
```

参数archive定义库的名称，files是库文件中包含的目标文件的清单，用空格分隔每个文件。

### 14.5.3 创建静态库文件

在使用ar命令之前，必须具有想要包含到库中的每个函数的目标文件。这可以像以往那样使用带有-o选项的as命令完成。

在创建库文件之前，应该判定库的命名约定。不同操作系统使用不同约定标识库文件。Linux操作系统使用下面的约定:

```
libx.a
```

其中x是库的名称。扩展名a标识这个文件是静态库文件。

创建新的存档文件和添加新文件的ar命令格式非常简单明了。下面的命令创建本章中迄今为止介绍过的汇编语言函数的存档文件:

```
$ ar r libchap14.a square.o cpuidfunc.o areafunc.o greater.o fpmathfunc.o
```

ar命令没有输出结果，只是创建库文件。可以使用命令行选项t显示包含在库中的文件:

```
$ ar t libchap14.a
cpuid.o
square.o
area.o
cpuidfunc.o
areafunc.o
greater.o
fpmathfunc.o
$
```

如果希望得到库文件的更加详细的清单，可以使用v选项:

```
$ ar tv libchap14.a
rw-r--r-- 501/501    592 Sep 21 19:03 2004 cpuid.o
rw-r--r-- 501/501    480 Sep 21 19:03 2004 square.o
rw-r--r-- 501/501    482 Sep 21 19:03 2004 area.o
rw-r--r-- 501/501    596 Sep 22 11:56 2004 cpuidfunc.o
rw-r--r-- 501/501    486 Sep 22 20:06 2004 areafunc.o
rw-r--r-- 501/501    940 Sep 22 18:00 2004 greater.o
rw-r--r-- 501/501   1104 Sep 21 20:55 2004 fpmathfunc.o
$
```

文件的时间戳标识汇编器创建文件的时间，而不是它们被添加到存档文件中的时间。

创建库文件之后，可以创建库的索引来帮助提高必须和库连接的其他程序的编译速度，这是个好主意。使用ranlib程序创建库的索引。索引存放在库文件内部。

运行ranlib程序时，不会显示非常令人激动的内容:

```
$ ranlib libchap14.a
$
```

可以使用nm程序显示存档文件的索引。nm程序用于显示目标文件的符号:

```
$ nm -s libchap14.a | more

Archive index:
output in cpuid.o
cpuid in cpuid.o
square in square.o
area in area.o
output in cpuidfunc.o
cpuidfunc in cpuidfunc.o
areafunc in areafunc.o
greater in greater.o
fpmath in fpmathfunc.o
```

nm命令显示每个函数的名称，并且显示哪个目标文件包含哪个函数。显示函数清单之后，

会生成每个目标文件中的标签的详细清单。

### 14.5.4 编译静态库

创建好静态库文件之后，可以使用它编译需要库中包含的任何函数的C程序：

```
$ gcc -o intttest inttest.c libchap14.a
$ gcc -o stringtest stringtest.c libchap14.a
$ gcc -o floattest floattest.c libchap14.a
$
```

使用库来编译程序不会影响生成的可执行文件的长度。通过比较使用单一函数目标文件和静态库文件的结果，可以看到这一点：

```
$ gcc -o inttest inttest.c square.o
$ ls -al inttest
-rwxr-xr-x    1 rich     rich        13838 Sep 22 16:36 inttest
$ gcc -o inttest inttest.c libchap14.a
$ ls -al inttest
-rwxr-xr-x    1 rich     rich        13838 Sep 22 16:37 inttest
$
```

生成的文件的长度完全相同，并且运行的结果也相同。

## 14.6 使用共享库

多亏Microsoft Windows的广泛使用，几乎熟悉计算机的任何人都知道共享库。共享库的Microsoft版本是声名狼藉的DLL文件。几乎每个人都遇到过升级的DLL文件破坏了应用程序的情况。

本节讨论什么是共享库，以及如何在Linux环境中使用汇编语言函数创建和使用它们。

### 14.6.1 什么是共享库

前一节讲过，当应用程序和静态库一起编译时，函数代码被编译到了应用程序中。这就是说应用程序所需的所有代码都在可执行程序文件中。

注意，这种方式有下面的缺陷：

- 如果在函数代码中改动某些内容，使用此函数的每个应用程序都必须使用新的版本重新编译。
- 使用相同函数的多个程序都必须包含相同的代码。这使小型的应用程序比它所需长度的要长，因为它必须包含用到的每个函数的所有代码。
- 运行在系统上的多个程序可能会使用相同函数，这意味着多次把相同的函数加载到内存中。

共享库试图解决这些问题。包含函数目标代码的单独文件被




图 14-6

加载到操作系统的通用区域中。当应用程序需要访问共享库中的函数时，操作系统自动地把函数代码加载到内存中，并且允许应用程序访问它。

如果另一个应用程序也需要使用此函数代码，操作系统允许它访问已经被加载到内存中的相同的函数代码。只有函数代码的一个拷贝被加载到了内存中，并且使用此函数代码的每个独立程序不需要把它加载到它们的内存空间中，或者它们的可执行文件中。图14-6显示这种情况。

如果需要对函数进行任何改动，唯一需要更新的文件只有单一的共享库文件。使用共享库的每个程序都会自动地使用函数的新版本。当然，当函数的改动改变了它原来的行为时会出现问题。这就是在Windows环境中导致使用共享库的应用程序被破坏的原因。(Linux提供一种创建版本号的方法，调用程序可以比较版本号。)

### 14.6.2 创建共享库

使用gcc编译器从目标文件创建共享库。在创建共享库之前，必须使用as对汇编语言函数进行汇编。和处理静态库一样，Linux具有用于共享库的命名约定:

```
libx.so
```

其中x是库的名称，扩展名.so表明这是共享库。

用于创建共享库的gcc命令行选项是-shared选项:

```
$ gcc -shared -o libchap14.so square.o cpuidfunc.o areafunc.o \
greater.o fpmathfunc.o
```

文件libchap14.so包含被导入的汇编函数的所有目标代码。为了标识版本信息，通常文件名追加有版本号，比如libchap14.so.1。和静态库不同，共享库中的目标代码不被编译到可执行程序中。

### 14.6.3 编译共享库

虽然共享库文件不被编译到C程序中，但是编译器仍然必须知道如何访问函数。使用-l选项加上共享库的名称（减去lib部分和.so扩展名），在编译命令行上导入共享库。在能够使用共享库之前，必须使用-L选项通知编译器在哪里查找它。和-L选项一起使用的参数是编译器查找共享库的其他位置，除了操作系统中定义的位置外（这在下一节中详细解释）。如果共享库位于和程序文件相同的目录中，可以使用句号表示相同的目录。

这使编译命令行如下:

```
$ gcc -o inttest -L. -lchap14 inttest.c
```

编译器创建不包含函数代码的可执行文件inttest。为了验证这一点，可以使用objdump程序对可执行文件inttest进行反汇编。main段不包含对函数square的调用。而是包含下面这一行:

```
80484c1:       e8 ea fe ff ff          call   80483b0 <_init+0x28>
```

不进行对函数square的调用。实际上，在反汇编后的可执行文件中的任何位置都找不到函数square的代码。被调用的是Linux动态加载器。这在第4章讲解在汇编语言程序中使用函数printf时讨论过。动态加载器用于在运行时自动地加载必须的共享库以便提供所需的函数。现在也可

以动态地加载汇编函数。

为了查看可执行文件依赖什么共享库，可以使用ldd命令：

```
$ ldd inttest
        libchap14.so => not found
        libc.so.6 => /lib/libc.so.6 (0x40027000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$
```

ldd命令显示执行可执行程序inttest需要共享库文件libchap14.so，并且在系统中没有找到它。这是个很大的问题。如果试图运行新的可执行文件，就会发生这样的结果：

```
$ ./inttest
./inttest: error while loading shared libraries: libchap14.so: cannot open shared
object file: No such file or directory
$
```

虽然程序员知道共享库和可执行文件位于相同的目录中，但是Linux动态加载器不知道。在能够使用共享库之前，还必须注意进行一项处理工作。

### 14.6.4 运行使用共享库的程序

动态加载器必须知道如何访问共享库libchap14.so。有两种方式通知它文件位于什么位置：

- LD_LIBRARY_PATH环境变量
- /ets/ld.so.conf文件

下面两节介绍这两种方法。

1. LD_LIBRARY_PATH环境变量

LD_LIBRARY_PATH环境变量是系统上的任何用户为当前进程的动态加载器添加路径的简单方式。它包含一个路径清单（以冒号分隔），动态加载器应该在ld.so.conf文件中列出的路径之外的这些位置查找库文件。使用LD_LIBRARY_PATH环境变量不需要任何专门的权限；只需设置它：

```
$ export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:."
$ ldd inttest
        libchap14.so => libchap14.so (0x40017000)
        libc.so.6 => /lib/libc.so.6 (0x40028000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ ./inttest
The square of 2 is 4
The square of 10 is 100
$
```

这个例子把句号追加到现有LD_LIBRARY_PATH值的结尾。句号在Linux中表示当前目录。这就是说，动态加载器将在当前工作目录中查找库文件libchap14.so。然后使用ldd命令，它会正确地查找库文件libchap14.so，并且程序将像预期那样运行。

2. /etc/ld.so.conf文件

文件ld.so.conf位于/etc目录中，它保存动态加载器在哪些目录中查找库的目录清单。在我的Debian系统中它是下面这样的：

```
$ cat /etc/ld.so.conf
/lib
/usr/lib
/usr/X11R6/lib
/usr/i486-linuxlibc1/lib

/usr/lib/libc5-compat
/lib/libc5-compat
$
```

这些目录是系统共享库文件应该在的位置，动态加载器会在这些位置查找它们。可以把库文件libchap14.so存放在这些目录的任何一个中，动态加载器将正确地找到它。但是，把系统库和应用程序库混合在一起并不是好做法。

最好为应用程序库创建单独的目录，并且把这个目录添加到文件ld.so.conf中。大多数Linux版本都包含/usr/local/lib目录，可以把应用程序共享库存储在其中。为了避免这个目录中的库文件太混乱，可以在这里为应用程序创建单独的子目录，并且把必须的应用程序库文件存储在这里。

把新的目录添加到文件ld.so.conf中之后，必须（以root用户身份）运行ldconfig命令以便更新文件ld.so.cache，动态加载器将使用它：

```
$ ldconfig
$ ldd inttest
        libchap14.so => /usr/local/lib/apps/libchap14.so (0x40026000)
        libc.so.6 => /lib/libc.so.6 (0x40028000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
$ ./inttest
The square of 2 is 4
The square of 10 is 100
$
```

ldconfig程序详细列出添加到ld.so.cache文件的目录中的库文件的清单。如果在运行ldconfig程序时，库文件没有被加载到其中一个目录，而是以后再添加，它就不会工作。每次添加新的库文件时，都必须运行ldconfig。

现在ldd命令表明它在/usr/local/lib/apps目录中找到了共享库。

准备好共享库之后，需要它的任何应用程序都将能够工作：

```
$ gcc -o stringtest -L/usr/local/lib/apps -lchap14 stringtest.c
$ ./stringtest
The CPUID is: 'GenuineIntel'
$
```

## 14.7　调试汇编函数

如果读者从头开始阅读本书，现在应该已经精通于使用GNU调试器（gdb）查找汇编语言应用程序的错误。但是，当试图调试在C或者C++程序中进行操作的汇编语言函数时，还要学习一些技巧。本节介绍调试汇编语言函数需要了解的一些事情。

### 14.7.1　调试C程序

使用gdb调试器调试C程序与使用它调试汇编语言程序非常类似。在编译C程序时必须使用-g

选项。为了获得Linux环境的更多专门信息，可以使用-gstabs选项：

```
$ gcc -gstabs -o inttest inttest.c square.s
```

这个命令生成包含C程序代码的调试信息的可执行文件。然后可以在调试器中启动程序，并且使用l命令列出源代码行：

```
$ gdb -q inttest
(gdb) l
1       /* inttest.c - An example of returning an integer value */
2       #include <stdio.h>
3
4       int main()
5       {
6               int i = 2;
7               int j = square(i);
8               printf("The square of %d is %d\n", i, j);
9
10              j = square(10);
(gdb) l
11              printf("The square of 10 is %d\n", j);
12              return 0;
13      }
(gdb)
```

调试C程序时，可以使用标签或者行号设置断点。为了在程序的开头设置断点，可以使用下面的命令：

```
(gdb) break *main
Breakpoint 1 at 0x8048460: file inttest.c, line 5.
(gdb)
```

现在可以启动程序，并且可以单步执行C语句。如果存在任何C变量，可以使用print命令显示它们：

```
(gdb) run
Starting program: /home/rich/palp/chap14/inttest

Breakpoint 1, main () at inttest.c:5
5       {
(gdb) s
main () at inttest.c:6
6               int i = 2;
(gdb) s
7               int j = square(i);
(gdb) s
8               printf("The square of %d is %d\n", i, j);
(gdb) print i
$1 = 2
(gdb) print j
$2 = 4
(gdb)
```

注意，在C语句中遇到汇编函数时，调试器把它当作单一语句对待。调试器执行到函数时，它会在单一执行步骤中完成它。这也许是你期望的结果，也许不是。

即使在编译中包含了汇编语言函数的源代码，调试信息不会添加到可执行程序中。下一节介绍如何改变这种情况。

### 14.7.2 调试汇编函数

如果希望调试器进入C程序内的汇编语言函数中，就必须单独地对函数进行汇编，并且在汇编器生成的目标代码文件中包含调试信息。然后可以把产生的目标文件和C程序编译在一起：

```
$ as -gstabs -o square.o square.s
$ gcc -gstabs -o inttest inttest.c square.o
```

现在在调试器中运行可执行程序，当逐步执行到汇编语言函数中时，将看到不同的结果：

```
$ gdb -q inttest
(gdb) break *main
Breakpoint 1 at 0x8048460: file inttest.c, line 5.
(gdb) run
Starting program: /home/rich/palp/chap14/inttest

Breakpoint 1, main () at inttest.c:5
5       {
(gdb) s
main () at inttest.c:6
6               int i = 2;
(gdb) s
7               int j = square(i);
(gdb) s
square () at square.s:5
5         pushl %ebp
Current language:  auto; currently asm
(gdb)
```

执行到汇编语言函数时，执行另一个步骤就会进入函数的源代码。注意，调试器表明它切换到了汇编语言格式。到达汇编语言函数的结尾时，执行下一个步骤会返回C程序：

```
(gdb) s
10        popl %ebp
(gdb) s
square () at square.s:11
11        ret
(gdb) s
main () at inttest.c:8
8               printf("The square of %d is %d\n", i, j);
Current language:  auto; currently c
(gdb)
```

如果在应用程序中有很多汇编语言函数，但是只希望单步调试特定的一个，可以使用next命令跳过函数：

```
(gdb) run
Starting program: /home/rich/palp/chap14/inttest

Breakpoint 1, main () at inttest.c:5
5       {
```

```
Current language:  auto; currently c
(gdb) s
main () at inttest.c:6
6               int i = 2;
(gdb) s
7               int j = square(i);
(gdb) n
8               printf("The square of %d is %d\n", i, j);
(gdb)
```

执行next命令时，调试器处理函数，但是不进入它。

因为汇编语言函数必须把调试信息汇编到其中，所以通常创建两个库是有帮助的：一个包含不带调试信息的汇编语言函数的目标文件，一个包含带有调试信息的目标文件。这使程序员可以选择把哪个版本编译到程序中，以便生成程序的开发型和产品型两个版本。

## 14.8 小结

本章介绍如何在C和C++程序中使用汇编语言函数。C或者C++程序调用的汇编函数必须采用特定的格式。C样式的函数格式需要把函数的所有输入值存放到堆栈中。函数必须保留调用程序使用的寄存器，比如EBX、EBP、ESP、EDI和ESI寄存器。这是通过在函数开头把寄存器的值压入堆栈，在函数返回调用程序之前把它们弹出堆栈来完成的。

函数从堆栈读取所有输入值，把调用程序要读取的整数和字符串指针结果存放到EAX寄存器中。如果返回浮点值，就把它存放到FPU的ST(0)寄存器中，由调用函数读取和清空它。如果C调用程序希望得到指针或者浮点返回值，就必须在源代码中编写汇编函数的原型，为编译器定义期望的返回值。

构建同时使用C程序和汇编语言函数的应用程序时，必须确保编译C程序时所有函数对编译器是可用的。这可以通过在编译器命令行中包含汇编语言函数的源代码文件来完成，或者通过单独对汇编语言函数进行汇编，然后在编译器命令行中包含目标代码文件来完成。

当应用程序使用很多汇编语言函数时，在编译器命令行中包含所有必须的文件会变得令人厌烦。替换的做法是，可以在单一位置创建包含所有汇编函数的目标文件的库文件。在Linux系统中可以使用两种类型的库文件。

静态库文件在一个文件中包含所有汇编语言函数的目标代码，这个文件可以编译到C主程序的可执行文件中。使用静态库文件创建的每个程序都包含它使用的汇编语言函数的完整目标代码。可执行文件可以独立运行，不需要操作系统的任何其他文件。

共享库文件也把多个汇编语言函数的目标文件组合到单一库文件中。共享库的不同之处在于目标代码没有被编译到C的主可执行文件中。而是当需要时，C可执行文件使用Linux系统的动态加载器程序从共享库文件调用目标代码。因为函数的目标代码被单独地加载到系统内存中，所以动态加载器可以使正在运行的其他程序重用相同的代码。

既然读者已经了解了如何在C程序中使用汇编语言函数，现在该开始进行实践了。下一章介绍如何使用C程序，以及确定如何使用汇编语言函数在应用程序的特定区域中帮助提高性能。

# 第15章 优 化 例 程

如果读者曾经使用C或者C++进行过专业的程序设计，选择这本书也许为了学习汇编语言来帮助优化应用程序。既然熟悉了汇编语言，那就开始分析自己的应用程序并把汇编语言知识投入实用。

但是，仅仅使用汇编语言代码替换C或者C++编写函数并不会必然使它们执行得更好。记住，GNU编译器已经把所有高级语言代码转换成了汇编语言，所以使用汇编语言编写函数仅仅意味着替编译器完成了这个任务。

为了真正地优化高级语言函数，需要编写比编译器生成的代码更好的汇编语言代码。这是非常大的挑战。可以使用若干优化技巧指示编译器生成汇编语言代码。开始研究编写自己的汇编语言函数之前，学习专家使用什么技术优化汇编语言代码，然后使用这些技术，甚至改进它们，是个好主意。

为了查看优化后的汇编语言代码，需要了解如何从编译器生成代码。本章首先讨论编译C或者C++程序时如何使用各种优化级别，以及在每个优化级别中利用了什么优化技术。之后，介绍优化过程的范例，使读者熟悉用于查看、修改和重新编译优化后的代码的技术。然后，剖析编译器使用的5种比较常用的优化技术，令读者学习优化技术如何工作，以及如何在自己的汇编语言编码工作中使用它们。

## 15.1 优化编译器代码

在C程序设计的早期，希望优化应用程序的程序员的一般工作是研究汇编语言的代码行，查找要优化的指令代码。现在，由于有了优化的编译器，大多数这类工作已经替我们完成了。GNU编译器也不例外。它包含一些选项，使得可以为代码编译针对的程序类型和处理器专门地指定最优化的调整方式。

编译器的-O选项系列提供GNU编译器的优化步骤。每个步骤都提供更高级别的优化。当前优化可用的有3个级别:

- -O: 提供基础级别的优化
- -O2: 提供更加高级的代码优化
- -O3: 提供最高级别的优化

不同优化级别使用的优化技术也可以单独地应用于代码。可以使用-f命令行选项引用每个单独的优化技术。-O选项和各种-f选项捆绑在一起构成单一选项。

本节介绍不同的-O优化级别，并且介绍每个级别包含什么-f选项。

### 15.1.1 编译器优化级别1

在优化的第一个级别执行基础代码优化。在这个级别试图执行9种单独的优化功能。我使用

“试图”这个词是因为不能保证任何优化功能肯定实现，编译器只是试图执行它们而已。下列清单介绍这个级别包含的-f优化功能：

- -fdefer-pop：这种优化技术与汇编语言代码在函数完成时如何进行操作有关。一般情况下，函数的输入值被存放到堆栈中并且被函数访问。函数返回时，输入值还在堆栈中。一般情况下，函数返回之后，输入值被立即弹出堆栈。

这个选项允许编译器跨越函数调用，使输入值累积在堆栈中。然后使用单一指令一次把所有这些累积的输入值删除（通常通过把堆栈指针改动为适当的值完成）。对于大多数操作，这是完全合法的，因为新的函数的输入值被存放到堆栈中旧的输入值的顶部。但是，这样使堆栈中的内容有些杂乱。

- -fmerge-constants：使用这种优化技术，编译器试图合并相同的常量。这一特性有时候会导致很长的编译时间，因为编译器必须分析C或者C++程序中用到的每个常量，并且相互比较它们。
- -fthread-jumps：这种优化技术与编译器如何处理汇编代码中的条件和非条件分支有关。在某些情况下，一条跳转指令可能转移到另一条分支语句。通过一连串跳转，编译器确定多个跳转之间的最终目标并且把第一个跳转重新定向到最终目标。
- -floop-optimize：通过优化如何生成汇编语言中的循环，编译器可以在很大程度上提高应用程序的性能。通常，程序由很多大型且复杂的循环构成。通过删除在循环内没有改变值的变量赋值操作，可以减少循环内执行的指令的数量，在很大程度上提高性能。此外，优化那些确定何时离开循环的条件分支，以便减少分支的影响。
- -fif-conversion：if-then语句是应用程序中仅次于循环的最消耗时间的部分。简单的if-then语句可能在最终的汇编语言代码中生产众多条件分支。通过减少或者删除条件分支，以及使用条件传送、设置标志和使用运算技巧替换它们，编译器可以减少if-then语句中花费的时间量。
- -fif-conversion2：这种技术结合更加高级的数学特性，减少实现if-then语句所需的条件分支。
- -fdelayed-branch：这种技术试图根据指令周期时间重新安排指令。它还试图把尽可能多的指令移动到条件分支之前，以便最充分地利用处理器指令缓存。
- -fguess-branch-probability：就像其名称所暗示的，这种技术试图确定条件分支最可能的结果，并且相应地移动指令，这和延迟分支（delayed-branch）技术类似。因为是在编译时预测代码的安排，所以使用这一选项两次编译相同的C或者C++代码很可能会产生不同的汇编语言源代码，这取决于在编译时编译器认为会使用哪些分支。

因为这个原因，很多程序员不喜欢采用这个特性，并且专门地使用-fno-guess-branch-probability选项关闭这个特性。

- -fcprop-registers：因为在函数中把寄存器分配给变量，所以编译器执行第二次检查以便减少调度依赖性（两个段要求使用相同的寄存器）并且删除不必要的寄存器复制操作。

### 15.1.2 编译器优化级别2

代码优化的第二个级别（-O2）结合了第一个级别的所有优化技术，再加上很多其他技术。

这些技术涉及更特定类型的代码，比如循环和条件分支。如果编译器生成的基础汇编语言代码没有利用这个级别中分析的代码类型，就不会执行附加的优化。下面的清单描述这个级别试图执行的附加的-f优化选项。

- -fforce-mem: 这种优化在任何指令使用变量之前，强制把存放在内存位置中的所有变量都复制到寄存器中。对于只涉及单一指令的变量，这样也许不会有很大的优化效果。但是，对于在很多指令（比如数学操作）中都涉及到的变量来说，这会是很显著的优化，因为和访问内存中的值相比，处理器访问寄存器中的值要快得多。
- -foptimize-sibling-calls: 这种技术处理相关的和/或递归的函数调用。通常，递归的函数调用可以被展开为一系列一般的指令，而不是使用分支。这样使处理器的指令缓存能够加载展开的指令并且处理它们，和指令保持为需要分支操作的单独函数调用相比，这样更快。
- -fstrength-reduce: 这种优化技术对循环执行优化并且删除迭代变量。迭代变量是捆绑到循环计数器的变量，比如使用变量、然后使用循环计数器变量执行数学操作的for-next循环。
- -fgcse: 这种技术对生成的所有汇编语言代码执行全局通用子表达式消除（Global Common Subexpression Elimination，gcse）例程。这些优化操作试图分析生成的汇编语言代码并且组合通用片断，消除冗余的代码段。

应该注意，如果代码使用计算性的goto，gcc指令推荐使用-fno-gcse选项。

- -fcse-follow-jumps: 这种特别的通用子表达式消除（Common Subexpression Elimination，cse）技术扫描跳转指令，查找程序中通过任何其他途径都不会到达的目标代码。这种情况最常见的例子就是if-then-else语句的else部分。
- -frerun-cse-after-loop: 这种技术在对任何循环已经进行过优化之后重新运行通用子表达式消除例程。这样确保在展开循环代码之后更进一步地优化循环代码。
- -fdelete-null-pointer-checks: 这种优化技术扫描生成的汇编语言代码，查找检查空指针的代码。编译器假设间接引用空指针将停止程序。如果在间接引用之后检查指针，它就不可能为空。
- -fexpensive-optimizations: 这种技术执行从编译时的角度来说代价高昂的各种优化技术，但是它可能对运行时的性能产生负面影响。
- -fregmove: 编译器试图重新分配MOV指令中使用的寄存器，并且将其作为其他指令的操作数，以便最大化捆绑的寄存器的数量。
- -fschedule-insns: 编译器将试图重新安排指令，以便消除等待数据的处理器。对于在进行浮点运算时有延迟的处理器来说，这使处理器在等待浮点结果时可以加载其他指令。
- -fsched-interblock: 这种技术使编译器能够跨越指令块调度指令。这可以非常灵活地移动指令以便使等待期间完成的工作最大化。
- -fcaller-saves: 这个选项指示编译器针对函数调用保存和恢复寄存器，使函数能够访问寄存器值，而且不必保存和恢复它们。如果调用多个函数，这样能够节省时间，因为只进行一次寄存器的保存和恢复操作，而不是在每个函数调用中都进行。
- -fpeephole2: 这个选项允许进行任何计算机特定的观察孔优化。
- -freorder-blocks: 这种优化技术允许重新安排指令块以便改进分支操作和代码局部性。

- -fstrict-aliasing: 这种技术强制实行高级语言的严格变量规则。对于C和C++程序来说，它确保不在数据类型之间共享变量。例如，整数变量不和单精度浮点变量使用相同的内存位置。
- -funit-at-a-time: 这种优化技术指示编译器在运行优化例程之前读取整个汇编语言代码。这使编译器可以重新安排不消耗大量时间的代码以便优化指令缓存。但是，这会在编译时花费相当多的内存，对于小型计算机可能是一个问题。
- -falign-functions: 这个选项用于使函数对准内存中特定边界的开始位置。大多数处理器按照页面读取内存，并且确保全部函数代码位于单一页面之内能够改进性能。如果函数跨越页面，为了完成函数就必须处理内存的另一个页面。
- -falign-loops: 和对准函数类似，在内存中的页面边界内对准包含多次被处理的代码的循环是有好处的。处理循环时，如果它包含在单一内存页面内，就不需要交换代码所需的页面。
- -fcrossjumping: 这是对跨越跳转的转换代码的处理，以便组合分散在程序各处的相同代码。这样可以减少代码的长度，但是也许不会对程序性能有直接影响。

### 15.1.3 编译器优化级别3

使用-O3选项访问编译器提供的最高级别的优化。它整合了第一和第二级别中的所有优化技术，还有一些非常专门的附加优化技术。再次说明，不能保证这个级别的优化将改进最终代码的性能。下面是这个级别包含的-f优化选项:

- -finline-functions: 这种优化技术不为函数创建单独的汇编语言代码，而是把函数代码包含在调用程序的代码中。对于多次被调用的函数来说，为每次函数调用复制函数代码。虽然这样对减少代码长度不利，但是通过最充分地利用指令缓存代码，而不是在每次函数调用时进行分支操作，可以提高性能。
- -fweb: 构建用于保存变量的伪寄存器网络。伪寄存器包含数据，就像它们是寄存器一样，但是可以使用各种其他优化技术进行优化，比如cse和loop优化技术。
- -fgcse-after-reload: 这种技术在完全重新加载生成的且优化后的汇编语言代码之后执行第二次gcse优化，帮助消除不同优化方式创建的任何冗余段。

## 15.2 创建优化的代码

可以使用gcc编译器从C和C++程序创建优化后的汇编语言代码。默认情况下，优化后的代码被编译为目标文件并且连接为可执行文件。如果希望分析优化后的代码，并且要学习它，甚至要改进它，就必须在编译代码之前截取生成的汇编语言代码。

本节介绍创建和查看从编译的C或者C++源代码生成的优化的汇编语言代码需要采取的步骤，以及如何把它汇编回一般的可执行应用程序。

### 15.2.1 生成汇编语言代码

优化汇编语言代码的第一个步骤是查看未经优化的版本，以便可以了解编译器对C或者C++源代码进行了什么操作。这涉及到使用GNU编译器的-S选项。

-S选项创建一个文件，这个文件包含从高级语言源代码生成的汇编语言代码。为了演示这种

情况，我们使用简单的C程序tempconv.c，它使用单一函数，把华氏温度转换为摄氏温度：

```
/* tempconv.c - An example for viewing assembly source code */
#include <stdio.h>

float convert(int deg)
{
        float result;
        result = (deg - 32.) / 1.8;
        return result;
}

int main()
{
        int i = 0;
        float result;
        printf("    Temperature Conversion Chart\n");
        printf("Fahrenheit       Celsius\n");
        for(i = 0; i < 230; i = i + 10)
        {
                result = convert(i);
                printf("  %d              %5.2f\n", i, result);
        }
        return 0;
}
```

程序tempconv.c没有特别之处。它使用单一函数进行温度转换的计算。程序的main部分使用单一循环对运算使用的一组输入值进行循环处理。

为了生成应用程序的基本的汇编语言代码，可以使用下面的命令：

```
$ gcc -S tempconv.c
```

这会创建文件tempconv.s，它包含编译器生成的汇编语言代码：

```
.file   "tempconv.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
        .align 8
.LC0:
        .long   0x0,0x40400000
        .align 8
.LC1:
        .long   0xcccccccd,0x3ffccccc
.text
        .align 16
.globl convert
        .type    convert,@function
convert:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        fildl   8(%ebp)
        fldl    .LC0
        fsubrp  %st, %st(1)
        fldl    .LC1
        fdivrp  %st, %st(1)
```

```
        fstps   -4(%ebp)
        flds    -4(%ebp)
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe1:
        .size    convert,.Lfe1-convert
                .section        .rodata
        .align 32
.LC3:
        .string "    Temperature Conversion Chart\n"
.LC4:
        .string "Fahrenheit       Celsius\n"
.LC5:
        .string "  %d             %5.2f\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $0, -4(%ebp)
        subl    $12, %esp
        pushl   $.LC3
        call    printf
        addl    $16, %esp
        subl    $12, %esp
        pushl   $.LC4
        call    printf
        addl    $16, %esp
        movl    $0, -4(%ebp)
        .p2align 4,,7
.L4:
        cmpl    $229, -4(%ebp)
        jle     .L7
        jmp     .L5
        .p2align 4,,7
.L7:
        subl    $12, %esp
        pushl   -4(%ebp)
        call    convert
        addl    $16, %esp
        fstps   -8(%ebp)
        flds    -8(%ebp)
        leal    -8(%esp), %esp
        fstpl   (%esp)
        pushl   -4(%ebp)
        pushl   $.LC5
        call    printf
        addl    $16, %esp
        leal    -4(%ebp), %eax
        addl    $10, (%eax)
        jmp     .L4
        .p2align 4,,7
.L5:
```

```
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

现在读者应该熟悉分析生成的应用程序汇编语言代码的工作了。函数convert()生成的代码应该是很容易理解的。首先，使用一般的C样式函数开头设置堆栈的基指针，并且为两个局部变量保留空间：

```
convert:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
```

接下来，从堆栈读取输入值并且把它加载到FPU堆栈中，把第一个数字型常量（32）加载到FPU堆栈中：

```
        fildl   8(%ebp)
        fldl    .LC0
```

接下来，执行减法操作，结果存储在FPU的寄存器ST(0)中：

```
fsubrp  %st, %st(1)
```

之后，把除数值加载到FPU堆栈中，然后执行除法步骤：

```
fldl    .LC1
fdivrp  %st, %st(1)
```

现在结果位于寄存器ST(0)中，函数返回时主程序将读取它。但是，编译器执行了一些奇怪的操作。首先它弹出FPU堆栈的值，把它加载到局部变量中，然后把值从局部变量加载回FPU堆栈中：

```
fstps   -4(%ebp)
flds    -4(%ebp)
```

这无疑是优化过程中可以删除的一些操作。

### 15.2.2 查看优化的代码

既然读者已经了解了编译器生成的基本汇编语言代码，现在该了解优化后的代码了。同样，为了查看汇编语言代码，必须使用编译器选项-S，还有-O3选项以便执行所有优化操作。默认情况下，生成的汇编语言文件将被存放在文件tempconv.s中。为了能够和生成的原始代码进行比较，可以使用-o选项声明不同的文件名称：

```
$ gcc -O3 -S -o tempconv2.s tempconv.c
```

这条命令生成汇编语言文件tempconv2.s，它包含优化后的代码。在优化后的文件中，函数convert()的代码如下：

```
convert:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %eax
        fildl   8(%ebp)
        fsubl   .LC0
        fdivl   .LC1
        fstps   -4(%ebp)
        flds    -4(%ebp)
        movl    %ebp, %esp
        popl    %ebp
        ret
```

注意其中有一些微妙的区别。没有使用FLD指令单独加载常量值。而是在数学指令中直接使用了它们。但是，优化过程没有删除从FPU堆栈弹出结果、然后把它压入回堆栈这些不必要的操作。为了进一步优化这个函数，可以删除最后的FSTPS和FLDS指令。

### 15.2.3 重新编译优化的代码

对编译器生成的汇编语言代码做出任何优化改动之后，必须使用它创建优化后的可执行文件。可以使用两种方法之一完成这个工作。第一种方法是使用一般的as和ld命令汇编和连接优化后的汇编语言代码文件（记住连接C库以便支持代码中的所有C函数）：

```
$ as -o tempconv.o tempconv2.s
$ ld -dynamic-linker /lib/ld-linux.so.2 -lc -o tempconv tempconv.s
```

第二种方法是使用gcc编译器处理汇编语言源代码文件：

```
$ gcc -o tempconv tempconv2.s
```

新的可执行文件包含使用编译器和自己的优化手段创建的完全优化了的代码。

## 15.3 优化技巧

不仅对于汇编语言函数，而且在一般的汇编语言程序中，编译器使用的很多优化手段都是很方便的。如果计划进行重要的汇编语言程序设计，那么熟悉用于优化汇编语言代码的一些比较常用的技巧是个好主意。

本节演示汇编语言中使用的5种最为常用的优化方法：

- 优化运算
- 优化变量
- 优化循环
- 优化条件分支
- 优化通用子表达式

下面几节演示这些优化技术，其中提供一个包含可以优化的C代码的范例C程序，生成未经优化的汇编语言代码，并且介绍如何使用编译器优化生成的汇编语言代码。

### 15.3.1 优化运算

处理方程式时，几乎总有机会可以简化一些运算。有时候，为了在使用涉及到的变量时显

示方程式流程，按照未经简化的形式把这些运算输入到C或者C++源代码中。还有些时候，缺乏经验的程序员输入了混乱的源代码。

在上述任何一种情况下，编译器都可以优化运算生成的汇编语言代码。本节演示如何优化汇编语言代码中的运算，以便帮助提高程序的性能。

1. 未经优化的运算

为了演示编译器如何优化运算，我们使用程序calctest.c:

```
/* calctest.c - An example of pre-calculating values */
#include <stdio.h>

int main()
{
        int a = 10;
        int b, c;
        a = a + 15;
        b = a + 200;
        c = a + b;
        printf("The result is %d\n", c);
        return 0;
}
```

这个范例程序演示使用变量和常量值执行简单的运算。使用前面的代码中定义的一个变量值计算每个值。使用编译器的-S选项为这个程序生成的默认汇编语言源代码如下:

```
.file   "calctest.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "The result is %d\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $10, -4(%ebp)
        leal    -4(%ebp), %eax
        addl    $15, (%eax)
        movl    -4(%ebp), %eax
        addl    $200, %eax
        movl    %eax, -8(%ebp)
        movl    -8(%ebp), %eax
        addl    -4(%ebp), %eax
        movl    %eax, -12(%ebp)
        subl    $8, %esp
        pushl   -12(%ebp)
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        movl    %ebp, %esp
```

```
        popl    %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

编译器生成的汇编语言代码使用标准C样式的函数格式创建程序（参见第14章“调用汇编库”）。编译器保留24个字节用于局部变量的存储，并且使用这个位置引用下面3个程序变量：

| 程序变量 | 堆栈存储位置 |
|---|---|
| a | -4 (%ebp) |
| b | -8 (%ebp) |
| c | -12 (%ebp) |

然后，编译器生成适当的汇编语言代码对所有局部变量执行定义好的运算：

```
movl    $10, -4(%ebp)
leal    -4(%ebp), %eax
addl    $15, (%eax)
movl    -4(%ebp), %eax
addl    $200, %eax
movl    %eax, -8(%ebp)
movl    -8(%ebp), %eax
addl    -4(%ebp), %eax
movl    %eax, -12(%ebp)
```

值10被传送到局部变量位置a。然后在ADD指令中使用间接寻址使变量a的内存位置加上值15。结果存放在EAX寄存器中，使它加上200，结果存放在变量堆栈位置b中。最后，变量b的值被加载到EAX寄存器中，和变量a的值相加。结果存放在变量堆栈位置c中。按照一般方式把结果压入堆栈，再把C函数printf使用的文本压入堆栈（注意编译器使用.string命令，它和.asciz命令相同）。

所有运算按照C代码期望的那样执行。但是，每次运行程序时，这是一种缓慢和单调的运算方式。下一节介绍允许编译器对代码进行优化时会发生什么情况。

2. 查看优化后的运算

不必每次都处理汇编语言代码以便执行运算，可以设置编译器对运算进行优化。编译器确定每个运算中使用的值，识别出程序运行过程中哪些值不会改变。

为了生成优化后的汇编语言代码，可以使用下面的命令：

```
$ gcc -O3 -S -o calctest2.s calctest.c
```

这会创建文件calctest2.s，它包含优化后的汇编语言源代码：

```
.file   "calctest.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "The result is %d\n"
.text
        .align 16
.globl main
```

```
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        pushl   $250
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    %ebp, %esp
        xorl    %eax, %eax
        popl    %ebp
        ret
.Lfe1:
.size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

这些就是全部。在代码长度上有很大不同！读者注意到的第一件事情也许是代码中没有使用局部变量。执行用来确定变量a、b和c的值的运算的所有代码都被删除了。因为程序中的其他位置没有使用任何变量值，唯一被计算出来的值就是最终结果。编译器计算最终值并且在汇编语言代码中供函数printf中直接使用。程序不需要在每次运行时计算这些值。

### 15.3.2 优化变量

优化应用程序的最明显的途径之一是控制汇编语言程序如何处理变量。处理变量有3种方式：

- 使用.data或者.bss段在内存中定义变量
- 使用EBP基指针在堆栈中定义局部变量
- 使用可用的寄存器保存变量值

本节介绍在程序中处理变量的不同技术，以及如何在汇编语言代码中优化它们。

#### 1. 使用未经优化的全局和局部变量

很多C和C++程序员并不了解在他们的程序中使用全局和局部变量的影响。对于大多数程序员来说，这仅仅是在程序中的什么位置声明变量的问题。但是，生成汇编语言代码时，如何处理变量有巨大区别。

程序vartest.c演示把C程序转换为汇编语言代码时如何创建变量：

```
/* vartest.c - An example of defining global and local C variables */
#include <stdio.h>

int global1 = 10;
float global2 = 20.25;

int main()
{
        int local1 = 100;
        float local2 = 200.25;
        int result1 = global1 + local1;
        float result2 = global2 + local2;
        printf("The results are %d and %f\n", result1, result2);
        return 0;
}
```

程序vartest.c定义两个全局变量（一个整数和一个浮点值），还有两个局部变量（同样是整数和浮点值）。为了从这段C代码创建基本的汇编语言代码，可以使用编译器的-S选项创建文件vartest.s:

```
.file   "vartest.c"
        .version        "01.01"
gcc2_compiled.:
.globl global1
.data
        .align 4
        .type    global1,@object
        .size    global1,4
global1:
        .long   10
.globl global2
        .align 4
        .type    global2,@object
        .size    global2,4
global2:
        .long   0x41a20000
                .section        .rodata
.LC1:
        .string "The results are %d and %f\n"
        .align 8
.LC0:
        .long   0xf01b866e,0x400921f9
.text
.align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    $100, -4(%ebp)
        movl    $0x43484000, -8(%ebp)
        movl    -4(%ebp), %eax
        addl    global1, %eax
        movl    %eax, -12(%ebp)
        flds    global2
        fadds   -8(%ebp)
        fstps   -16(%ebp)
        flds    -16(%ebp)
        leal    -8(%esp), %esp
        fstpl   (%esp)
        pushl   -12(%ebp)
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

读者应该能够理解这些汇编代码，辨别出发生了什么情况。在.data段中把两个全局变量定义为长类型值（这样创建4字节的值）：

```
global1:
        .long   10
global2:
        .long   0x41a20000
```

编译器把浮点值20.25转换为等同的十六进制单精度浮点数据类型，并且使用.long命令直接存储它。

辨别局部变量存储在什么位置稍微困难一些。和C样式的函数代码（参见第14章）一样，通过从堆栈指针ESP减去保留空间的总量，在堆栈中为局部变量保留空间。然后把两个局部变量存储在为局部变量保留的前两个位置中：

```
pushl   %ebp
movl    %esp, %ebp
subl    $24, %esp
movl    $100, -4(%ebp)
movl    $0x43484000, -8(%ebp)
```

在堆栈中保留24字节的空间。前面两个位置用于保存局部变量。第一个局部变量存储在-4（%ebp）位置中。第二个局部变量存储在-8（%ebp）位置中。

把局部变量存储到堆栈中之后，从这些位置引用变量的值进行整数运算：

```
movl    -4(%ebp), %eax
addl    global1, %eax
movl    %eax, -12(%ebp)
```

局部变量被传送到EAX寄存器中，在ADD指令中它和全局变量相加。结果存储在堆栈中保留的第三个局部变量位置中。

浮点运算利用FPU寄存器堆栈：

```
flds    global2
fadds   -8(%ebp)
fstps   -16(%ebp)
flds    -16(%ebp)
```

第一个全局变量被加载到寄存器ST(0)中，然后使用FADD指令把它和局部变量相加。和前面tempconv.s程序中看到的一样，编译器再次从FPU堆栈弹出结果，存放到局部变量中，然后把它加载回FPU堆栈中以便让主程序获得：

```
leal    -8(%esp), %esp
fstpl   (%esp)
pushl   -12(%ebp)
pushl   $.LC0
call    printf
```

LEA指令用于从堆栈中清除局部变量，从FPU堆栈获得浮点结果并且存放到程序堆栈中，再把函数printf()使用的显示字符串存放到堆栈中。

2. 经过优化的全局和局部变量

使用相同的vartest.c程序，可以看到如何使用下面的GNU编译器的命令行选项优化全局和局

部变量:

```
$ gcc -O3 -S -o vartest2.s vartest.c
```

这将生成汇编语言文件vartest2.s，它包含程序优化后的代码:

```
.file   "vartest.c"
        .version        "01.01"
gcc2_compiled.:
.globl global1
.data
        .align 4
        .type    global1,@object
        .size    global1,4
global1:
        .long   10
.globl global2
        .align 4
        .type    global2,@object
        .size    global2,4
global2:
        .long   0x41a20000
                .section        .rodata
.LC1:
        .string "The results are %d and %f\n"
        .align 4
.LC0:
        .long   0x43484000
.text
.align 16
.globl main
        .type    main,@function
main:
        flds    global2
        pushl   %ebp
        movl    global1, %eax
        fadds   .LC0
        movl    %esp, %ebp
        addl    $100, %eax
        subl    $16, %esp
        fstpl   (%esp)
        pushl   %eax
        pushl   $.LC1
        call    printf
        addl    $16, %esp
        movl    %ebp, %esp
        xorl    %eax, %eax
        popl    %ebp
        ret
.Lfe1:
        .size    main,.Lfe1-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

读者应该注意到的第一件事情是，没有使用我们通常看到的完全C样式的函数开头，编译器在开头指令中混合了其他指令:

```
flds    global2
```

```
pushl   %ebp
movl    global1, %eax
fadds   .LC0
movl    %esp, %ebp
addl    $100, %eax
```

main段中的第一条指令是FLD指令，它把全局值加载到FPU中。编译器知道把值存储到FPU寄存器的过程中具有固有的延迟，而且知道在处理FPU指令时，处理器可以继续执行其他指令。把全局值存储到FPU堆栈中时，把EBP寄存器的值存储到程序堆栈中。存储EBP值之后，把整数全局值存储到可用的寄存器中，以便帮助提高对值的访问速度。

读者注意到的下一件事情也许是没有在堆栈中定义局部变量。没有在堆栈中创建局部变量，而是直接把局部变量的值传送到了数学指令中（FADD和ADD）。和使用内存中的全局变量相比，这样将很大程度地节省处理器时间。把存储在全局内存位置中的变量加载到FPU中将需要额外的时间。还要注意，这次优化器确定可以把浮点结果从FPU堆栈中删除并且直接使用，而不是弹出它然后再次压入它。

### 15.3.3 优化循环

程序循环可能是应用程序中最为消耗时间的部分之一。经常可以优化为了模拟do-while和for-next循环而生成的汇编语言代码以便加快执行速度。编译器试图使用若干优化技术减少循环需要的代码数量。

本节介绍如何优化程序中一般的for-next循环以便最小化循环中花费的时间。

1. 一般的for-next循环代码

在第6章“控制执行流程”中演示过如何在汇编语言代码中实现for-next循环的基础知识。在汇编语言中实现for-next循环的伪代码如下：

```
for:
   <condition to evaluate for loop counter value>
   jxx forcode     ; jump to the code of the condition is true
   jmp end         ; jump to the end if the condition is false
forcode:
  < for loop code to execute>
  <increment for loop counter>
   jmp for         ; go back to the start of the For statement
end:
```

这段代码需要用3个分支语句来实现for-next循环。汇编语言代码中的分支对性能的影响可能是灾难性的，因为它使预加载到指令缓存中的指令完全失去了利用价值。

优化循环的最好方式是要么消除它们，要么至少试图简化它们，使用的手段是循环展开。展开循环需要非常多的额外代码（不使用分支重复执行代码，而是按照循环应该处理的次数编写代码）。虽然这样的代码不能减少程序的长度，但是确保指令预取缓存能够完成其工作并且预先加载指令。性能提高也许能够超过程序更大而导致的代价，但也许不能够做到。

2. 查看循环代码

为了演示如何优化简单的循环，我们把程序sums.c转换为汇编代码。首先，下面是C程序代码：

```
/* sums.c - An example of optimizing for-next loops */
#include <stdio.h>

int sums(int i)
{
        int j, sum = 0;
        for(j = 1; j <= i; j++)
                sum = sum + j;
        return sum;
}

int main()
{
        int i = 10;
        printf("Value: %d   Sum: %d\n", i, sums(i));
        return 0;
}
```

程序sums.c包含单一函数，它确定从1开始到输入值的连续整数的总和。使用for-next循环确定总和。编译器生成的实现这个程序的未经优化的汇编语言代码如下：

```
.file   "sums.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 16
.globl sums
        .type    sums,@function
sums:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $0, -8(%ebp)
        movl    $1, -4(%ebp)
        .p2align 4,,7
.L3:
        movl    -4(%ebp), %eax
        cmpl    8(%ebp), %eax
        jle     .L6
        jmp     .L4
        .p2align 4,,7
.L6:
        movl    -4(%ebp), %eax
        leal    -8(%ebp), %edx
        addl    %eax, (%edx)
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L3
        .p2align 4,,7
.L4:
        movl    -8(%ebp), %eax
        movl    %eax, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe1:
        .size    sums,.Lfe1-sums
```

```
                .section        .rodata
.LC0:
        .string "Value: %d   Sum: %d\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $10, -4(%ebp)
        subl    $4, %esp
        subl    $8, %esp
        pushl   -4(%ebp)
        call    sums
        addl    $12, %esp
        movl    %eax, %eax
        pushl   %eax
        pushl   -4(%ebp)
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

在汇编语言函数sums中实现for-next循环。它包含复杂的条件分支和比较指令。

首先，创建两个局部变量保存循环计数（存储在-4（%ebp）位置，设置为1）和正在计算的总和值（存储在-8（%ebp）位置，设置为0）。代码这样实现循环：

```
.p2align 4,,7
.L3:
        movl    -4(%ebp), %eax
        cmpl    8(%ebp), %eax
        jle     .L6
        jmp     .L4
        .p2align 4,,7
.L6:
        movl    -4(%ebp), %eax
        leal    -8(%ebp), %edx
        addl    %eax, (%edx)
        leal    -4(%ebp), %eax
        incl    (%eax)
        jmp     .L3
```

循环计数被加载到EAX寄存器中并且和函数的输入值（存储在一般的8（%ebp）位置）进行比较。如果循环计数小于或者等于输入值，代码就跳转到使循环计数与总和值相加的段。加法操作完成之后，循环计数递增1，代码跳转回去，再次加载循环计数值。注意循环计数值已经被加载到了EAX寄存器中，所以每次循环迭代都重新加载它是浪费循环内的执行时间。

3. 优化for-next循环

现在我们可以查看编译器如何优化代码，但是要使用编译器的命令行选项-O3:

```
$ gcc -O3 -S -o sums2.s sums.c
```

这会创建下面的汇编语言代码:

```
.file   "sums.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "Value: %d   Sum: %d\n"
.text
        .align 16
.globl sums
        .type    sums,@function
sums:
        pushl   %ebp
        movl    $1, %edx
        movl    %esp, %ebp
        xorl    %eax, %eax
        movl    8(%ebp), %ecx
        cmpl    %ecx, %edx
        jg      .L30
        .p2align 4,,7
.L21:
        addl    %edx, %eax
        incl    %edx
        cmpl    %ecx, %edx
        jle     .L21
.L30:
        popl    %ebp
        ret
.Lfe1:
        .size    sums,.Lfe1-sums
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        xorl    %edx, %edx
        movl    %esp, %ebp
        movl    $1, %eax
        subl    $12, %esp
        .p2align 4,,7
.L26:
        addl    %eax, %edx
        incl    %eax
        cmpl    $10, %eax
        jle     .L26
        pushl   %edx
        pushl   $10
        pushl   $.LC0
        call    printf
        addl    $16, %esp
```

```
        movl    %ebp, %esp
        xorl    %eax, %eax
        popl    %ebp
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

注意，优化操作发生在函数sums的代码中。不使用局部变量作为循环计数和总和值，而是使用寄存器值（EDX寄存器用于循环计数，EAX寄存器用于正在计算的总和值）。输入值也被立即加载到了ECX寄存器中，所以现在所有数学指令都对存储在寄存器中的值执行。

把这些值加载到寄存器中之后，第一条比较指令确保循环计数不大于最终值。如果循环计数已经大于最终值，就不需要循环迭代，并且代码会跳过循环：

```
cmpl    %ecx, %edx
jg      .L30
```

接下来，创建极为紧密的循环来执行计算总和的操作：

```
.L21:
        addl    %edx, %eax
        incl    %edx
        cmpl    %ecx, %edx
        jle     .L21
```

同样，所有值都包含在寄存器中，所以在循环内不需要访问内存。还要注意，循环完成时结果已经包含在EAX寄存器中，以便返回调用程序，无需把它从内存传送给寄存器。

### 15.3.4 优化条件分支

优化汇编语言程序的另一个问题是优化条件分支。和循环类似，条件分支能够破坏预加载到指令缓存中的指令，如果采用了没有预测到的分支，就会导致处理器进行额外的工作。

条件分支的主要应用之一是在if-then类型的语句中。C和C++程序包含众多的if-then语句用于评估程序代码中的条件，并且根据这些条件处理数据，这种情况并不少见。

本节演示GNU编译器如何把if-then语句转换为汇编语言代码，并且演示编译器如何优化if-then类型的语句。了解编译器如何优化汇编语言程序中的条件分支对程序员自己的汇编语言编码技术有很大帮助。

1. 普通的if-then代码

第6章也讲解过在汇编语言代码中实现if-then逻辑的一般方案：

```
if:
   <condition to evaluate>
   jxx  else        ; jump to the else part if the condition is false
<code to implement the "then" statements>
jmp end             ;jump to the end
else:
   < code to implement the "else" statements>
end:
```

标准的if-then模板利用一个条件跳转和一个非条件跳转实现代码逻辑。下一节介绍编译器如

何从实际C程序中的if-then代码实现这种逻辑。

2. 查看if-then代码

为了查看编译器生成的汇编语言代码，我们首先需要一个使用if-then语句的应用程序。程序condtest.c提供一个简单的例子，它在函数中使用if-then-else语句，根据两个输入值返回特定值：

```
/* condtest.c - An example of optimizing if-then code */
#include <stdio.h>

int conditiontest(int test1, int test2)
{
        int result;
        if (test1 > test2)
        {
                result = test1;
        } else if (test1 < test2)
        {
                result = test2;
        }else
        {
                result = 0;
        }
        return result;
}

int main()
{
        int data1 = 10;
        int data2 = 30;
        printf("The result is %d\n", conditiontest(data1, data2));
        return 0;
}
```

这个例子执行简单的if-then-else条件测试，返回两个输入值中的大值。如果两个输入值相等，则返回0。为这个程序生成的未经优化的汇编语言代码显示在文件condtest.s中：

```
.file   "condtest.c"
        .version        "01.01"
gcc2_compiled.:
.text
        .align 16
.globl conditiontest
        .type    conditiontest,@function
conditiontest:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $4, %esp
        movl    8(%ebp), %eax
        cmpl    12(%ebp), %eax
        jle     .L3
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        jmp     .L4
        .p2align 4,,7
.L3:
        movl    8(%ebp), %eax
```

```
        cmpl    12(%ebp), %eax
        jge     .L5
        movl    12(%ebp), %eax
        movl    %eax, -4(%ebp)
        jmp     .L4
        .p2align 4,,7
.L5:
        movl    $0, -4(%ebp)
.L4:
        movl    -4(%ebp), %eax
        movl    %eax, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe1:
        .size    conditiontest,.Lfe1-conditiontest
                .section        .rodata
.LC0:
        .string "The result is %d\n"
.text
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $10, -4(%ebp)
        movl    $30, -8(%ebp)
        subl    $8, %esp
        pushl   -8(%ebp)
        pushl   -4(%ebp)
        call    conditiontest
        addl    $8, %esp
        movl    %eax, %eax
        pushl   %eax
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    $0, %eax
        movl    %ebp, %esp
        popl    %ebp
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

注意函数conditiontest()中实现的典型if-then-else代码:

```
        movl    8(%ebp), %eax
        cmpl    12(%ebp), %eax
        jle     .L3
        movl    8(%ebp), %eax
        movl    %eax, -4(%ebp)
        jmp     .L4
        .p2align 4,,7
.L3:
        movl    8(%ebp), %eax
```

```
        cmpl    12(%ebp), %eax
        jge     .L5
        movl    12(%ebp), %eax
        movl    %eax, -4(%ebp)
        jmp     .L4
        .p2align 4,,7
.L5:
        movl    $0, -4(%ebp)
.L4:
        movl    -4(%ebp), %eax
```

从堆栈把第一个输入值加载到EAX寄存器中，在CMP指令中直接访问堆栈获得第二个输入值，把它和第一个输入值进行比较。如果第一个值小于或者等于第二个值，程序就跳转到逻辑的第一个else部分。如果不是，if-then语句的原始条件就为真，并且执行then部分。把第一个输入值加载到第一个局部变量值中并且跳转到代码末尾。

if-then-else逻辑的else部分在标签.L3处开始。注意在这个标签之前，使用了新的指令.p2align指令。

在第5章“传送数据”中已经见过了.align指令。使用它确保数据元素对准适当的内存边界，以便提高把值加载到寄存器（特别是FPU寄存器）中的速度。指令.p2align和它类似，但是对如何对准元素提供更多的控制。

指令.p2align的格式如下：

```
.p2align number, value, max
```

参数number定义地址中必须为0的低位0位的位数。在这个例子中，4这个值说明内存地址必须是16的倍数（低4位必须为0）。

参数value定义将在填充字节中使用的值。如果跳过这个值（就像这个例子这样），则在填充字节中使用0。

参数max定义对准指令应该跳过的最大字节数量。在这个例子中，这个值为7，它表示对准下一段应该跳过的字节数量不应该超过7字节，否则就会忽略指令.p2align。

标签.L3之后的代码，通过比较堆栈中的第一个输入值和第二个输入值实现第二个if-then-else语句。如果if-then条件为假，代码就跳转到第二个else部分，位于标签.L5的位置。如果条件为真，就把第二个输入值加载到第一个局部变量值中，并且跳转到代码末尾。

第二个else部分直接把0加载到第一个局部变量值中。代码末尾把第一个局部变量值加载到EAX寄存器中，以便把它返回给调用程序。

3. 经过优化的if-then代码

同样，可以使用编译器选项-O3获得程序优化后的版本。文件condtest2.s显示优化后的代码：

```
.file   "condtest.c"
        .version        "01.01"
gcc2_compiled.:
                .section        .rodata
.LC0:
        .string "The result is %d\n"
.text
        .align 16
.globl conditiontest
```

```
        .type    conditiontest,@function
conditiontest:
        pushl   %ebp
        movl    %esp, %ebp
        movl    8(%ebp), %edx
        movl    12(%ebp), %ecx
        movl    %edx, %eax
        cmpl    %ecx, %edx
        jg      .L19
        xorl    %eax, %eax
        cmpl    %ecx, %edx
        setge   %al
        decl    %eax
        andl    %ecx, %eax
.L19:
        popl    %ebp
        ret
.Lfe1:
        .size    conditiontest,.Lfe1-conditiontest
        .align 16
.globl main
        .type    main,@function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        pushl   $30
        pushl   $.LC0
        call    printf
        addl    $16, %esp
        movl    %ebp, %esp
        xorl    %eax, %eax
        popl    %ebp
        ret
.Lfe2:
        .size    main,.Lfe2-main
        .ident  "GCC: (GNU) 2.96 20000731 (Linux-Mandrake 8.0 2.96-0.48mdk)"
```

优化后的代码在很大程度上减少了实现双重if-then-else语句需要的代码数量：

```
        movl    8(%ebp), %edx
        movl    12(%ebp), %ecx
        movl    %edx, %eax
        cmpl    %ecx, %edx
        jg      .L19
        xorl    %eax, %eax
        cmpl    %ecx, %edx
        setge   %al
        decl    %eax
        andl    %ecx, %eax
.L19:
```

第一个优化动作把两个输入值加载到寄存器中。如果第一个输入值（包含在EAX寄存器中）是最大值，就会立即知道它是返回值，并且函数可以返回调用程序。

如果EAX中的值不大于第二个输入值（包含在ECX寄存器中），就使用一些数学技巧确定最终的返回值。在这个例子中，编译器知道第二个输入值必然等于或者大于第一个输入值。再次

使用CMP指令比较这两个值。这次使用的条件指令是SETGE指令，如果第一个值大于或者等于第二个值，这条指令就把AL寄存器的值设置为1。因为我们知道，在这个部分中第一个值不可能大于第二个值（因为如果是这样，就已经满足第一个条件分支了），所以我们知道两个值肯定相等，AL寄存器的值被设置为1。如果第一个值小于第二个值，AL寄存器就被设置为0。

下一个部分是使用技巧的部分。EAX寄存器递减1，所以如果第一个值等于第二个值，那么现在EAX寄存器的值就为0；如果它小于第二个值，那么现在EAX寄存器的值就为-1，或者说为全1。下一条指令使用AND处理第二个输入值，因为EAX寄存器要么为0，要么为全1，所以如果两个值相等，最终值则为0，如果第二个值更大，最终值则为第二个值。试图优化复杂的汇编语言代码时，这种“从暗箱外考虑”的方式是至关重要的。

### 15.3.5 通用子表达式消除

编译器执行的更加高级的优化技术之一是通用子表达式消除（common subexpression elimination，cse）。编译器必须扫描整个汇编语言代码，查找通用表达式。找到经常使用的表达式时，就只需计算表达式一次；之后，可以在用到这个表达式的所有其他位置使用结果值。

本节通过一个简单的例子介绍如何在汇编语言代码中使用cse。

1. 程序范例

程序csetest.c演示在C程序中编写了通用表达式的简单情况：

```
/* csetest.c - An example of implementing cse optimization */
#include <stdio.h>

void funct1(int a, int b)
{
   int c = a * b;
   int d = (a * b) / 5;
   int e = 500 / (a * b);
   printf("The results are c=%d  d=%d  e=%d\n", c, d, e);
}

int main()
{
   int a = 10;
   int b = 25;
   funct1(a, b);
   funct1(20, 10);
   return 0;
}
```

函数funct1()包含3个方程式，每个方程式都包含相同的表达式a*b。未经优化的函数代码版本在遇到这3个方程式时执行3次乘法：

```
funct1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    8(%ebp), %eax
        imull   12(%ebp), %eax
        movl    %eax, -4(%ebp)
```

```
        movl    8(%ebp), %eax
        movl    %eax, %ecx
        imull   12(%ebp), %ecx
        movl    $1717986919, %eax
        imull   %ecx
        sarl    %edx
        movl    %ecx, %eax
        sarl    $31, %eax
        subl    %eax, %edx
        movl    %edx, %eax
        movl    %eax, -8(%ebp)
        movl    8(%ebp), %eax
        movl    %eax, %edx
        imull   12(%ebp), %edx
        movl    $500, %eax
        movl    %edx, %ecx
        cltd
        idivl   %ecx
        movl    %eax, -12(%ebp)
        movl    -12(%ebp), %eax
        movl    %eax, 12(%esp)
        movl    -8(%ebp), %eax
        movl    %eax, 8(%esp)
        movl    -4(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        leave
        ret
```

注意在3个实例中，都使用IMUL指令使第二个输入值和已经被加载到寄存器中的第一个输入值相乘。尽管编译器在这个回合没有优化通用表达式，但是可以注意到编译器执行了有趣的数学优化。为了获得变量d的值，编译器必须使乘法的结果除以5。它没有使用DIV指令执行除法操作，而是使用SAR指令向右移位寄存器的值以便执行除法。

因为SAR指令只能除以2的幂，所以需要执行附加的减法，实现除以5的除法。尽管使用了更多指令，但是这种方法仍然比使用单一DIV指令要快。

2. 使用cse进行优化

使用优化技术编译相同的程序之后，可以查看函数funct1()的汇编语言代码出现了什么改动：

```
funct1:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    %ebx, -8(%ebp)
        movl    12(%ebp), %ecx
        movl    8(%ebp), %ebx
        movl    %esi, -4(%ebp)
        movl    $1717986919, %esi
        imull   %ebx, %ecx
        movl    $.LC0, (%esp)
        movl    %ecx, %eax
        imull   %esi
        movl    $500, %eax
        movl    %ecx, 4(%esp)
```

```
movl    %edx, %ebx
movl    %ecx, %edx
sarl    $31, %edx
sarl    %ebx
subl    %edx, %ebx
movl    %ebx, 8(%esp)
cltd
idivl   %ecx
movl    %eax, 12(%esp)
call    printf
movl    -8(%ebp), %ebx
movl    -4(%ebp), %esi
movl    %ebp, %esp
popl    %ebp
ret
```

在优化后的代码中，把第一个输入值加载到EBX寄存器中，把第二个输入值加载到ECX寄存器中，然后使用IMUL指令使这两个值相乘，这样只执行一次乘法操作。结果值保存在ECX寄存器中，在其他两个方程式中使用这个结果值表示a*b表达式，不必重新计算这个值。

## 15.4 小结

仅仅使用手动编码的汇编语言代码替换C或者C++函数并不一定能够提高程序的性能。必须能够改进编译器生成的汇编语言代码，识别出可以增进性能的任何方式。如果使用现在具有优化功能的编译器，那么进一步优化不是容易的任务，但是可以做到。

优化汇编语言例程的第一个步骤是查看编译器如何创建它们。本章介绍如何使用GNU编译器的-O系列选项从C程序创建优化后的汇编语言代码。还要使用-S选项创建要分析的汇编语言源代码文件。通过查看GNU编译器如何优化汇编语言的特定函数，可以学习如何把这些代码应用到自己的汇编语言应用程序中。

本章介绍用于优化汇编语言函数的5种常用的优化方法。第1种方法是优化运算。大多数高级语言程序利用数学方程式处理数据。数据处理经常涉及到使用程序中已知的值进行运算。编译器不是盲目地把常量传送到内存位置或者寄存器中用于运算，而是分析运算得出的最终结果并且试图创建捷径。常常可以在数学指令中直接使用常量值，而不使用内存位置。这样节省了把值加载到内存中和从内存中卸载值的处理时间。

本章讨论的第2种方法是优化处理变量的方式。在C和C++程序中，可以把变量声明为局部的或者全局的。全局变量定义在汇编语言代码的.data段中，存储在内存位置中。默认情况下，在汇编语言代码中，局部变量定义在堆栈中。编译器试图优化代码时，会在可能的情况下把局部变量传送到寄存器中。这样可以在很大程度上提高应用程序的性能，因为访问寄存器比访问内存中的值要快很多。当然，对于使用众多变量的大型应用程序，没有足够的寄存器可以利用。为了尽量提高性能使其最大化，编译器会在必要时在寄存器和堆栈之间交换变量。

本章讨论的第3种方法涉及循环。循环是程序优化过程中最需要技巧的部分之一，但是如果适当地进行优化，也可以对性能产生巨大影响。奔腾计算机的主要特性之一是在处理器需要执行指令之前，预取指令并且把指令加载到指令缓存的能力。不幸的是，分支会干扰这个处理过程，导致刷新指令缓存并且从头开始进行缓存。优化的目的是尽可能地消除循环。对于短的迭

代循环，可以展开代码，复制循环每次迭代中的代码，而不是使用循环。虽然这样会增加程序的长度，但是不使用循环使处理器能够持续地执行代码，而节省时间的收益经常超过长度增加的代价。

第4种优化方法和循环的问题类似，但是优化对象是条件分支。和循环一样，如果采取不同的执行路径，条件分支就会导致处理器重新加载指令缓存。解决这个问题的方案是让最频繁用到的路径直接跟在跳转指令之后（跳转到较少用到的路径）。另外，当根据另一个值把变量设置为不同值时，经常可以使用数学技巧消除额外的跳转指令。

第5种方法经常是最为令人混淆的。它处理消除汇编语言程序内使用的通用表达式的问题。处理数学方程式的程序经常包含在多个方程式中用到的表达式，比如两个值相乘。编译器能够检测到汇编语言代码中多次使用前面计算过的值的情况，并且把计算的结果存储在一个位置中（通常是寄存器），以后可以从这里方便地访问这个结果。在运行时，这样可以确保程序只执行一次计算，而不是每次用到表达式时都需要计算。

虽然很多程序员从提高性能的角度考虑汇编语言代码，但是汇编语言的功能比这要多得多。下一章讨论如何使用汇编语言程序执行高级的文件访问。当程序员考虑文件访问时，他们通常考虑借助于高级语言函数，但是汇编语言程序可以像C和C++程序一样容易地访问文件。

# 第16章 使 用 文 件

在很多场合中，应用程序需要存储数据以便以后使用，或者从配置文件读取配置信息。为了处理这些功能，汇编语言程序必须能够和UNIX系统上的文件进行交互。

本章介绍在汇编语言程序中如何处理文件，包括把数据写入文件和从文件读取数据。使用两种基本方法从汇编语言程序访问文件。一种方法使用标准C函数。如果读者曾经使用C或者C++进行过程序设计，应该熟悉这些函数——fopen()、read()和write()。接下来，C文件的I/O函数使用Linux系统调用（在第12章“使用Linux系统调用”中讲解过）访问文件。在汇编语言程序设计中，可以绕过C函数调用，直接访问内核提供的Linux文件I/O系统调用。本章就介绍这种方法。

本章首先简要讲解UNIX系统如何处理文件。接下来，介绍如何把数据写入文件，指定适当的权限和访问类型。之后，介绍如何从文件读取数据，还有如何处理数据以及写入另一个文件。最后，讨论使用内存映射文件的主题。这一特性使得可以把整个文件读取到内存中，处理和修改内存映射文件中的数据，然后把数据写入原始文件。

## 16.1 文件处理顺序

和使用C和C++进行程序设计一样，在汇编语言程序中处理数据文件时必须使用特定的顺序。图16-1显示必须采取的事件顺序。

这些动作的每一个——打开、读取、写入和关闭——都通过Linux系统调用执行。第12章“使用Linux系统调用”中介绍过，代表系统调用的系统调用值被加载到EAX寄存器中，并且把所有参数值加载到其他通用寄存器中。

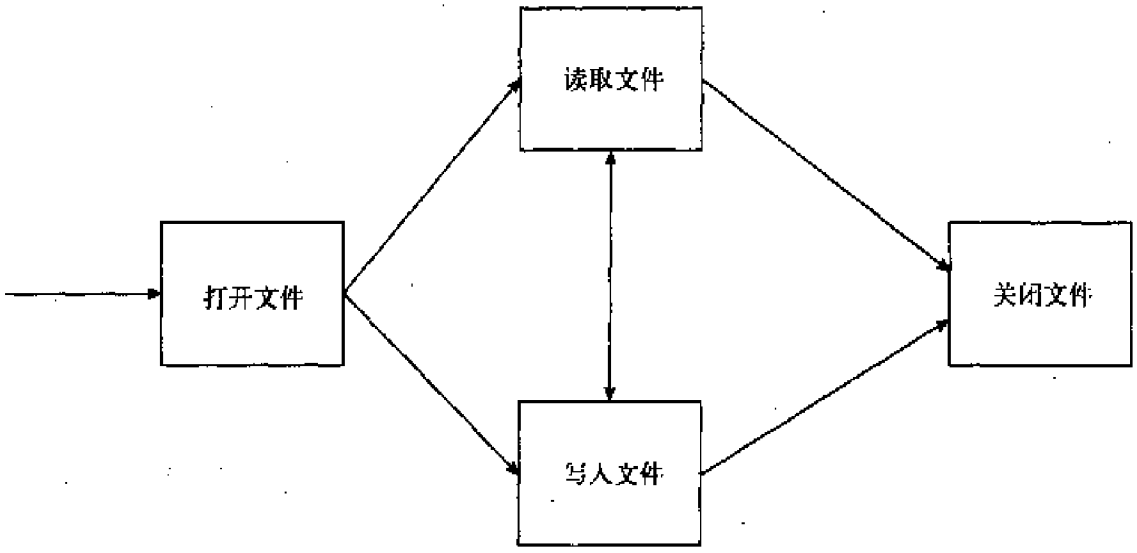



图 16-1

下表列出文件处理系统调用的Linux系统调用值。

| 系统调用 | 值 | 描　述 |
|---|---|---|
| 打开 | 5 | 打开要访问的文件并且创建指向文件的文件句柄 |
| 读取 | 3 | 使用文件句柄读取打开的文件 |
| 写入 | 4 | 使用文件句柄写入文件 |
| 关闭 | 6 | 关闭文件并且删除文件句柄 |

每个文件系统调用都有其自己的一组输入值，在发出系统调用之前必须配置它们。下面几节详细介绍每个系统调用，并且给出在汇编语言程序中使用它们的例子。

## 16.2 打开和关闭文件

如果熟悉C或者C++语言中文件的打开和关闭，在汇编语言中使用Linux系统调用打开和关闭文件是完全相同的。C函数open()使用系统调用open，把必须的参数传递给系统调用。系统调用open返回一个文件句柄，使用它标识打开的文件以便其他文件处理系统调用使用。完成文件处理之后，使用系统调用close关闭文件：

系统调用open的格式如下：

```
int open (const char *pathname, int flags, mode_t mode);
```

pathname是包含所有子目录的完整的以空字符结尾的文件路径。如果只指定文件名，就假设文件和运行的应用程序在相同的目录中。输入值flags确定对于此文件允许的文件访问，如果是创建文件，输入值mode确定UNIX权限设置。

汇编语言系统调用open从下面的寄存器读取必须的参数：

- EAX：包含系统调用值5。
- EBX：包含以空字符结尾的文件名字符串的开始位置的内存地址。
- ECX：包含表示需要文件的访问类型的标志的整数值。
- EDX：如果是创建新文件，则包含表示UNIX权限的整数值。

为文件定义的访问类型和UNIX权限是至关重要的。下面几节介绍这两个值的可能设置。

### 16.2.1 访问类型

对文件的所有open请求都必须声明用于打开文件的访问类型。如果读者曾经使用C函数open()打开文件，可能熟悉使用预先定义的常量，比如O_RDONLY或者O_RDWR。不幸的是，在汇编语言程序中没有定义这些常量。必须使用它们代表的数字值或者自己定义常量。这些常量的数字值通常表示为八进制值。常量的数字值显示在下表中。

| C常量 | 数字值 | 描　述 |
|---|---|---|
| O_RDONLY | 00 | 打开文件，用于只读访问 |
| O_WRONLY | 01 | 打开文件，用于只写访问 |
| O_RDWR | 02 | 打开文件，用于读写访问 |
| O_CREAT | 0100 | 如果文件不存在，就创建文件 |
| O_EXCL | 0200 | 和O_CREAT一起使用时，如果文件存在，就不打开它 |

（续）

| C常量 | 数字值 | 描 述 |
|---|---|---|
| O_TRUNC | 01000 | 如果文件存在并且按照写模式打开，则把文件长度截断为0 |
| O_APPEND | 02000 | 把数据追加到文件的结尾 |
| O_NONBLOCK | 04000 | 按照非块模式打开文件 |
| O_SYNC | 010000 | 按照同步模式打开文件（同时只允许一个写入操作） |
| O_ASYNC | 020000 | 按照异步模式打开文件（同时允许多个写入操作） |

可以将文件访问类型组合起来以便启用多种访问特性。例如，如果希望创建文件并且打开它用于读写访问，可以使用下面的指令：

```
movl $0102, %ecx
```

这把O_CREATE的值0100和O_RDWR的值02组合在一起。常量值中高位部分的0很重要。这表示这个值是八进制格式的。如果使用常量$102，就会得到错误结果，因为编译器会使用十进制值102。

如果希望把新的数据追加到已有的文件，可以使用这条指令：

```
movl $02002, %ecx
```

它把O_APPEND的值02000和O_RDWR的值02组合在一起，并且没有设置O_CREAT值。如果文件不存在，就不会创建它。

### 16.2.2 UNIX权限

设置UNIX权限经常会导致复杂情况。操作时必须谨慎，以便确保为文件访问设置适当的权限。标准UNIX权限针对3种类别的用户进行设置：

- 文件的所有者
- 文件的默认组
- 系统上的其他所有用户

这3种类别的每一种都被分配了文件的特定权限。使用3位表示每个类别的访问：

- 读取位
- 写入位
- 执行位

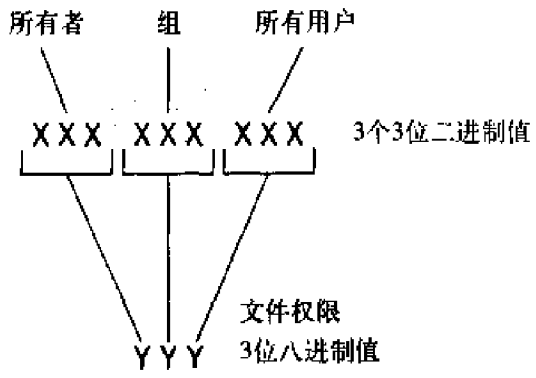



图 16-2

对准这3位形成每个类别一个3位值，如图16-2所示。

如图16-2所示，可以使用一个八进制值表示3位，表明设置了哪些位。可以将值组合起来生成各种访问级别，如下表所示。

| 权限位 | 值 | 访 问 |
|---|---|---|
| 001 | 1 | 执行权限 |
| 010 | 2 | 写入权限 |
| 011 | 3 | 执行和写入权限 |

（续）

| 权限位 | 值 | 访　问 |
|---|---|---|
| 100 | 4 | 读取权限 |
| 101 | 5 | 执行和读取权限 |
| 110 | 6 | 执行和写入权限 |
| 111 | 7 | 执行、写入和读取权限 |

将文件模式属性组合起来，形成单一的3位八进制数字，表示所有者、组和所有用户的权限。这个3位的八进制数字在系统调用open中定义，还有高位部分的0，使它成为八进制值。

指令

```
movl $0644, %edx
```

把八进制值644赋值给EDX寄存器。数字6表示所有者对文件有读/写权限。中间的数字4表示文件分配的组对文件有只读权限，第二个数字4表示系统上其他所有用户对文件有只读权限。

注意，对此有一个说明：Linux系统为登录到系统上的每个用户分配一个umask值。umask值把默认的权限分配给这个用户创建的文件。创建的文件的最终权限如下：

```
file privs = privs & ~umask
```

umask值被反转，并且和系统调用open中请求的权限进行AND操作。可以使用umask命令查看分配给用户帐户的umask值：

```
$ umask
022
$
```

分配给这个用户的umask是八进制值022。如果执行的系统调用open请求使用0666权限（所有用户都可以读/写）来创建文件，分配给创建的文件的最终权限就如下：

```
final privileges = privs & ~umask
                 = 666 & ~022
                 = 666 & 755
                 = 644
```

没有修改为所有者请求的权限，但是通过umask值拒绝了为组和其他所有用户请求的写入权限。这是系统上经常使用的umask值。它防止无意中授予文件的写入权限。

如果确实希望授予所有用户对文件的写入权限，就必须改变umask值，或者使用chmod命令手动地改动文件的权限。

### 16.2.3 打开文件代码

系统调用的最后一部分是声明要打开的文件名。如果没有使用路径，就假设文件和运行的可执行程序位于相同的目录中。

文件名必须声明为空字符结尾的字符串。可以使用.asciz声明完成这个工作：

```
.section .data
filename:
   .asciz "output.txt"
```

```
.
.
.section .text
.
.
   movl $filename, %ebx
```

因为EBX寄存器包含字符串位置的内存地址，所以必须在变量名称前面使用美元符号以便获得字符串位置的内存地址。

经常使用的另外一种方法是允许文件名被声明为程序的命令行参数。第11章“使用函数”中介绍过如何从程序堆栈访问命令行参数。位置8（%ebp）包含指向第一个命令行参数的内存位置的指针。这正是我们需要的文件名；因此，可以使用指令

```
movl %esp, %ebp
.
.
movl 8(%ebp), %ebx
```

把作为第一个命令行参数列出的文件名传送到系统调用open的函数调用中。在本章后面的例子中将详细进行讲解。

把所有部分组合在一起，可以使用下面这样的范例代码片断打开文件：

```
movl $5, %eax
movl $filename, %ebx
movl $0102, %ecx
movl $0644, %edx
int $0x80
test %eax, %eax
js badfile
```

这个代码片断使用O_CREAT和O_WRONLY访问，使用UNIX权限0644（所有者可以进行读/写访问，其他任何用户可以进行读取访问）打开文件（通过数据变量filename声明）。

系统调用返回时，EAX寄存器包含一个带符号整数值。这个值要么是非负值，它是打开文件的文件句柄；要么是负值，它是表示为什么不能打开文件的错误代码。因为错误代码是负值，所以可以使用JS条件分支指令，通过检查符号位进行检查。

通常，会希望把打开文件的文件句柄存储在某个位置以便以后在程序中使用。因为文件句柄是简单的32位带符号整数值，所以可以把它保存在可以存储任何其他整数值的任何位置中：

- .data段中定义的内存位置
- .bss段中定义的内存位置
- 堆栈，在局部变量段中
- 寄存器

在操作系统认为文件被打开的整个过程中，文件句柄值将保持为合法的。使用系统调用close关闭文件时，文件句柄变为非法的，并且不能使用它访问文件。

### 16.2.4 打开错误返回代码

如果系统调用open返回错误代码，可以把它和系统的头文件errno.h中定义的errno值进行比

较（汇编语言的errno.h文件通常位于系统上的/usr/include/asm目录中）。下表介绍可能返回的一些比较常见的错误代码。

| 错误名称 | 错误值 | 描　述 |
|---|---|---|
| EPERM | 1 | 操作不被许可 |
| ENOENT | 2 | 没有此文件 |
| EBADF | 9 | 坏文件句柄数字 |
| EACCES | 13 | 权限被拒绝 |
| EFAULT | 14 | 坏文件地址 |
| EBUSY | 16 | 设备或者资源忙 |
| EEXIST | 17 | 文件存在 |
| EISDIR | 21 | 是目录 |
| EMFILE | 24 | 过多打开文件 |
| EFBIG | 27 | 文件过大 |
| EROFS | 30 | 只读文件系统 |
| ENAMERTOOLONG | 36 | 文件名过长 |

记住错误代码被返回为负的数字，所以值将是errno表中显示值的负值。

### 16.2.5 关闭文件

完成对文件的使用之后，正确的操作是关闭它；否则，就有可能造成文件破坏。系统调用close使用单一输入参数——要关闭的已打开文件的文件句柄。参数存放在EBX寄存器中：

```
movl filehandle, %ebx
movl $6, %eax
int $0x80
```

系统调用close把错误代码返回到EAX寄存器中，使用errno.h错误代码。如果成功地关闭了文件，EAX寄存器将包含0。

## 16.3 写入文件

打开文件之后，可以使用系统调用write写入它。在本书中，当我们把数据写入控制台显示时（比如在第4章“汇编语言程序范例”中），已经看到过使用系统调用write的范例。控制台STDOUT的文件句柄永远都是默认值1。在这个特殊的情况中，不需要打开文件句柄，直接写入它即可。对于写入文件而不是写入控制台STDOUT的文件句柄的情况，使用从文件的系统调用open返回的文件句柄。默认的STDERR文件句柄的行为与STDOUT类似（它使用文件句柄2）。

### 16.3.1 简单的写入范例

在下面的程序cpuidfile.s中演示写入文件的简单汇编语言程序：

```
# cpuidfile.s - An example of writing data to a file
.section .data

filename:
   .asciz "cpuid.txt"
```

```
output:
   .asciz "The processor Vendor ID is 'xxxxxxxxxxxx'\n"
.section .bss
   .lcomm filehandle, 4
.section .text
.globl _start
_start:
   movl $0, %eax
   cpuid
   movl $output, %edi
   movl %ebx, 28(%edi)
   movl %edx, 32(%edi)
   movl %ecx, 36(%edi)

   movl $5, %eax
   movl $filename, %ebx
   movl $01101, %ecx
   movl $0644, %edx
   int $0x80
   test %eax, %eax
   js badfile
   movl %eax, filehandle

   movl $4, %eax
   movl filehandle, %ebx
   movl $output, %ecx
   movl $42, %edx
   int $0x80
   test %eax, %eax
   js badfile

   movl $6, %eax
   movl filehandle, %ebx
   int $0x80

badfile:
   movl %eax, %ebx
   movl $1, %eax
   int $0x80
```

程序cpuidfile.s利用第4章“汇编语言程序范例”中的原始程序cpuid.s，并且把它改为将输出数据写入文本文件。它使用O_TRUNC、O_CREAT和O_WRONLY文件访问模式打开（或者创建）文件cpuid.txt用于写入。如果文件已经存在，就把文件截断（清除其内容），因为没有包含O_APPEND访问模式。UNIX权限被设置为0644，允许用户读取和写入文件，但是只允许系统上的其他用户进行读取访问。发出系统调用open之后，检查返回值以便确保它不包含错误代码。如果它是合法的文件句柄，就把它存储到标签为filehandle的内存位置中。

然后把文件句柄加载到EBX寄存器中，使用系统调用write对它进行操作，把从CPUID指令得到的输出值写入到文件中。

通过简单地汇编、连接和运行程序，可以检查这个程序：

```
$ as -o cpuidfile.o cpuidfile.s
$ ld -o cpuidfile cpuidfile.o
$ ./cpuidfile
```

```
$ ls -al cpuid.txt
-rw-r--r--    1 rich     rich           42 Oct  6 09:18 cpuid.txt
[rich@test2 chap16]$ cat cpuid.txt
The processor Vendor ID is 'GenuineIntel'
$
```

程序成功地创建了文件，并且按照预期把输出字符串存放到了文件中。通过重新运行程序并且检查输出文件，可以检查文件访问设置：

```
$ ./cpuidfile
$ ls -al cpuid.txt
-rw-r--r--    1 rich     rich           42 Oct  6 09:20 cpuid.txt
$ cat cpuid.txt
The processor Vendor ID is 'GenuineIntel'
$
```

第二次运行的输出清除了原始数据，从文件的开头位置开始把新的数据写入已有文件。这使新的数据替换已有文件中包含的数据（当然，看不出这种情况，因为使用的数据是相同的）。通过使用编辑器把其他文本行添加到输出文件cpuid.txt中，然后重新运行程序cpuidfile，可以检查这种情况。

### 16.3.2 改变文件访问模式

程序生成的新文件替换了已有文件中的数据。如果想要把新的数据追加到文件中的已有数据之后，那么可以改变文件访问模式值。使用下面的代码替换系统调用open的代码：

```
movl $5, %eax
movl $filename, %ebx
movl $02101, %ecx
movl $0644, %edx
int $0x80
test %eax, %eax
js badfile
movl %eax, filehandle
```

注意，只改变了ECX寄存器中设置的值，现在添加了O_APPEND访问模式值，还有O_CREAT和O_WRONLY值。重新汇编和连接新的代码之后，可以进行检查：

```
$ ./cpuidfile2
$ ls -al cpuid.txt
-rw-r--r--    1 rich     rich           84 Oct  6 09:26 cpuid.txt
$ cat cpuid.txt
The processor Vendor ID is 'GenuineIntel'
The processor Vendor ID is 'GenuineIntel'
$
```

正如我们期望的，程序的新版本把输出的测试字符串追加到了文件cpuid.txt中已有的内容之后。

### 16.3.3 处理文件错误

在Linux环境中，有很多原因会造成访问文件的尝试失败。其他进程可能锁住文件，用户可能意外地删除文件，甚至没有经验的系统管理员可能把错误的权限分配给文件。必须确保汇编

语言代码准备好处理失败的情况。

在cpuidfile.s的例子中显示，最好检测文件访问系统调用返回的代码。除了把它作为系统调用exit的返回代码之外，这个例子没有做任何处理，但是在产品型的程序中，希望显示与遇到的特定错误代码相关的某种类型的错误消息。

测试错误代码处理的简单方式是强制产生错误的条件。对于这个例子来说，我把输出文件改变为只读文件，然后再次运行程序：

```
$ chmod 444 cpuid.txt
$ ls -al cpuid.txt
-r--r--r--    1 rich     rich           84 Oct  6 09:46 cpuid.txt
$ ./cpuidfile
$ echo $?
243
$
```

输出文件cpuid.txt的UNIX文件模式被设置为444，表示文件的所有者只能进行只读访问，系统上的任何其他用户也一样。运行程序cpuidfile并且检查返回代码时，会返回错误代码。

因为UNIX程序的返回代码是无符号整数，所以返回代码值被解释为正值。可以通过从256减去这个值确定真正的错误代码，其结果为13。从文件errno.h，可以确定错误代码13是EACCES错误，它表示不具有写入文件的权限。

## 16.4 读取文件

要处理的下一个步骤是读取文件中包含的数据。使用系统调用read完成这个功能。系统调用read的UNIX man页如下：

```
ssize_t read (int fd, void *buf, size_t count);
```

系统调用read使用3个输入值，生成单一输出值。3个输入值如下：

- 从其读取数据的文件的文件句柄
- 存放读出数据的缓冲区位置
- 试图从文件读取的字节数量

返回值表示系统调用从文件实际读出的字节数量。数据类型ssize_t和数据类型size_t类似，但是它是带符号整数值。这是因为如果发生错误，read函数可能返回负值。如果系统调用read返回0，则表示已经到达了文件的末尾。

能够读取的字节可能少于输入值中指定的字节数量。这可能是由于到达了文件的末尾造成的，也可能是发出调用read时数据不可用造成的。

当然，在发出系统调用read之前要把输入值存放在寄存器中。系统调用read使用的寄存器如下：

- EAX：read的系统调用值（3）
- EBX：打开文件的文件句柄
- ECX：数据缓冲区的内存位置
- EDX：要读取的字节数量的整数值

可以看到，系统调用read需要打开文件的文件句柄，所以在能够使用系统调用read之前，必须使用系统调用open打开文件。确保使用read访问模式设置（使用O_RDONLY或者O_RDWR模

式值）打开文件，这很重要。这里有一个例外：和STDOUT的情况类似，可以使用STDIN文件句柄（值为0）从标准输入设备（通常是键盘）读取数据，而不必打开它。

### 16.4.1 简单的读取范例

程序readtest1.s演示从前面介绍的程序cpuidfile.s创建的文件cpuid.txt读取数据：

```
# readtest1.s - An example of reading data from a file
.section .bss
   .lcomm buffer, 42
   .lcomm filehandle, 4
.section .text
.globl _start
_start:
   nop
   movl %esp, %ebp
   movl $5, %eax
   movl 8(%ebp), %ebx
   movl $00, %ecx
   movl $0444, %edx
   int $0x80
   test %eax, %eax
   js badfile
   movl %eax, filehandle

   movl $3, %eax
   movl filehandle, %ebx
   movl $buffer, %ecx
   movl $42, %edx
   int $0x80
   test %eax, %eax
   js badfile

   movl $4, %eax
   movl $1, %ebx
   movl $buffer, %ecx
   movl $42, %edx
   int $0x80
   test %eax, %eax
   js badfile

   movl $6, %eax
   movl filehandle, %ebx
   int $0x80

badfile:
   movl %eax, %ebx
   movl $1, %eax
   int $0x80
```

程序readtest1.s在程序的第一个命令行参数中指定要打开的文件名（位于8（%ebp）位置）。因为这个程序仅仅读取数据，所以在O_RDONLY模式下打开文件。接下来，执行系统调用read，指向.bss段中的标签buffer指定的缓冲区区域。指示系统调用read读取42字节的数据（程序cpuidfile.s的输出，包括新行字符）。

下一段使用系统调用write和文件句柄STDOUT显示存储在缓冲区区域中的数据（同样指定42字节的数据）。最后，使用系统调用close关闭指向文件cpuid.txt的文件句柄，然后使用系统调用exit退出程序。

运行程序时，记住在命令行中包含文件名cpuid.txt，因为它是程序将打开的文件名的位置：

```
$ ./readtest1 cpuid.txt
The processor Vendor ID is 'GenuineIntel'
$
```

程序像我们期望的那样执行。如果忘记在命令行中包含文件名，就不会显示任何内容，并且生成错误代码。可以使用shell值$?查看错误代码：

```
$ ./readtest1
$ echo $?
242
$
```

它相当于错误代码256-242=14，即错误EFAULT，说明8（%ebp）位置没有指定文件名地址。

当然，在真正的程序中，首先检查参数数量以便确保可以从命令行读取某些数据，这样的做法要好得多，而不是让open函数发出错误消息。

### 16.4.2 更加复杂的读取范例

程序readtest1.s是一个有些不切实际的例子。它指定需要从文件读取的数据的确切数量。更加常见的情况是，无法确切地知道需要读取多少数据。替换的做法是，必须循环遍历整个文件，直到到达文件末尾。

可以使用这种方法是因为系统调用read的工作方式。每次使用系统调用read时，在打开的文件中使用文件指针表示从文件读取的数据的最后一个字节。如果使用另一次系统调用read，它就从紧跟在文件指针位置后面的字节开始读取。读取完成时，把文件指针转移到读出的最后一个字节。这样持续操作，直到到达到文件的末尾。

系统调用read到达文件的末尾的标志是读取操作返回0值。必须在每次迭代时检查返回值，检查错误或者0值的情况。如果返回值为0，就说明已经到达了文件的末尾。

程序readtest2.s演示这种技术：

```
# readtest2.s - A more complicated example of reading data from a file
.section .bss
   .lcomm buffer, 10
   .lcomm filehandle, 4

.section .text
.globl _start
_start:
   nop
   movl %esp, %ebp
   movl $5, %eax
   movl 8(%ebp), %ebx
   movl $00, %ecx
   movl $0444, %edx
   int $0x80
```

```
    test %eax, %eax
    js badfile
    movl %eax, filehandle

read_loop:
    movl $3, %eax
    movl filehandle, %ebx
    movl $buffer, %ecx
    movl $10, %edx
    int $0x80
    test %eax, %eax
    jz done
    js done
    movl %eax, %edx
    movl $4, %eax
    movl $1, %ebx
    movl $buffer, %ecx
    int $0x80
    test %eax, %eax
    js badfile
    jmp read_loop

done:
    movl $6, %eax
    movl filehandle, %ebx
    int $0x80

badfile:
    movl %eax, %ebx
    movl $1, %eax
    int $0x80
```

程序readtest2.s同样从第一个命令行参数获得要打开的文件名。打开文件，开始循环，从文件读取10字节的数据块，把它们存放到缓冲区位置中。然后使用系统调用write把缓冲区位置写入到显示。系统调用read返回0值时，我们就知道已经到达了文件末尾，并且可以关闭文件句柄。

现在可以使用程序readtest2.s显示任何长度的文件：

```
$ ./readtest2 cpuid.txt
The processor Vendor ID is 'GenuineIntel'
The processor Vendor ID is 'GenuineIntel'
$
```

程序readtest2显示文件cpuid.txt的全部内容。可以使用任何文本文件来测试这个程序，包括它自己的源代码文件：

```
$ ./readtest2 readtest2.s
# readtest2.s - A more complicated example of reading data from a file
.section .bss
    .lcomm buffer, 10
    .lcomm filehandle, 4
    .lcomm size, 4
.section .text
.
.
.
badfile:
```

```
    movl %eax, %ebx
    movl $1, %eax
    int $0x80
$
```

整个源代码文件被显示出来，当到达文件末尾时，程序readtest2正确地停止了。

我们使用10字节的块长度是为了显示循环操作的效果。在产品型的应用程序中，希望使用更大的块长度使执行系统调用read的次数更少，以便改进应用程序的性能。

## 16.5 读取、处理和写入数据

当处理从文件读出的数据时，经常希望把数据写回到文件中。在readtest和readtest2的例子中，输出文件是控制台显示STDOUT。它可以很容易地变成另一个文件的文件句柄。

系统调用read使用文件指针跟踪已经从文件读出了什么数据，和它一样，系统调用write保持指向已经写入到文件的最后数据的文件指针。可以把数据块写入输出文件，然后使用另一次系统调用write把更多数据写到原始的数据块之后。

程序readtest3.s演示从文件读取数据、处理数据，然后把数据写入另一个文件中：

```
# readtest3.s - An example of modifying data read from a file and outputting it
.section .bss
    .lcomm buffer, 10
    .lcomm infilehandle, 4
    .lcomm outfilehandle, 4
    .lcomm size, 4
.section .text
.globl _start
_start:
    # open input file, specified by the first command line param
    movl %esp, %ebp
    movl $5, %eax
    movl 8(%ebp), %ebx
    movl $00, %ecx
    movl $0444, %edx
    int $0x80
    test %eax, %eax
    js badfile
    movl %eax, infilehandle

    # open an output file, specified by the second command line param
    movl $5, %eax
    movl 12(%ebp), %ebx
    movl $01101, %ecx
    movl $0644, %edx
    int $0x80
    test %eax, %eax
    js badfile
    movl %eax, outfilehandle

    # read one buffer's worth of data from input file
read_loop:
    movl $3, %eax
    movl infilehandle, %ebx
    movl $buffer, %ecx
```

```
    movl $10, %edx
    int $0x80
    test %eax, %eax
    jz done
    js badfile
    movl %eax, size

    # send the buffer data to the conversion function
    pushl $buffer
    pushl size
    call convert
    addl $8, %esp

    # write the converted data buffer to the output file
    movl $4, %eax
    movl outfilehandle, %ebx
    movl $buffer, %ecx
    movl size, %edx
    int $0x80
    test %eax, %eax
    js badfile
    jmp read_loop
done:
    # close the output file
    movl $6, %eax
    movl outfilehandle, %ebx
    int $0x80

    # close the input file
    movl $6, %eax
    movl infilehandle, %ebx
    int $0x80
badfile:
    movl %eax, %ebx
    movl $1, %eax
    int $0x80

# convert lower case letters to upper case
.type convert, @function
convert:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %esi
    movl %esi, %edi
    movl 8(%ebp), %ecx
convert_loop:
    lodsb
    cmpb $0x61, %al
    jl skip
    cmpb $0x7a, %al
    jg skip
    subb $0x20, %al
skip:
    stosb
    loop convert_loop
    movl %ebp, %esp
    popl %ebp
    ret
```

如果读者从头阅读本书，也许能够认出程序readtest3.s中使用的汇编语言函数convert。它基本上就是第10章“处理字符串”中使用的程序convert.s。它使用LODS指令，一次一字节地把字符串加载到EAX寄存器中。ESI寄存器指向源字符串的内存位置，EDI寄存器指向目标字符串的内存位置（它们被设置为指向相同的缓冲区区域）。

函数convert检查每个字节，查看它是否是小写的ASCII字符。如果是，就使这个值减去32，把它转换为大写ASCII字符。使用STOS指令把字符串存储回目标内存位置中。

这个函数被重新编写为新的汇编语言函数，从堆栈获得缓冲区和缓冲区长度变量。在调用这个函数之前，主程序把这些值压入堆栈。函数返回时，转换后的字节存放在相同的缓冲区区域中，然后把它们写入到输出文件。

程序readtest3.s使用两个命令行参数。第一个是要转换的输入文件的文件名。第二个是输出文件的文件名。下面是使用汇编后的程序的例子：

```
$ ./readtest3 cpuid.txt test.txt
$ cat test.txt
THE PROCESSOR VENDOR ID IS 'GENUINEINTEL'
THE PROCESSOR VENDOR ID IS 'GENUINEINTEL'
$ cat cpuid.txt
The processor Vendor ID is 'GenuineIntel'
The processor Vendor ID is 'GenuineIntel'
$
```

输出文件包含输入文件被转换后的数据。因为程序readfile3使用和程序readfile2相同的块读取方法，所以可以使用这个程序转换希望转换的任何长度的文本文件。

## 16.6 内存映射文件

如果运气实在不好，也许会遇到程序readtest3的一个问题。如果错误地试图使用相同的文件名作为输入和输出文件名，就会发生这样的结果：

```
$ ./readtest3 cpuid.txt cpuid.txt
$ echo $?
0
$ cat cpuid.txt
$ ls -l cpuid.txt
-rw-r--r--    1 rich     rich            0 Oct  6 15:00 cpuid.txt
$
```

程序运行了并且没有生成错误代码，但是输出文件是空的。文件存在，但是其中没有输出的数据。

问题在于系统不能在读取文件的同时把数据写入同一个文件。很多应用程序都必须更新文件。如果需要在应用程序中使用这个功能，有几种不同方式去解决它。一种方法称为内存映射文件（memory-mapped files）。

### 16.6.1 什么是内存映射文件

内存映射文件使用系统调用mmap把部分文件映射到系统的内存中。文件（或者部分文件）

被存放到内存中之后，程序可以使用标准内存访问指令访问内存位置，并且如果必须的话，可以修改它们。可以在多个进程之间共享内存位置，这使多个程序可以同时更新同一个文件。图16-3显示这种情况。

把文件加载到内存中之后，操作系统控制哪些用户可以对内存区域进行什么类型的访问。内存映射文件的内容可以写回原始文件，使用内存映射文件的内容替换原始文件的内容。这是更新几乎任何长度的文件（最大到系统的虚拟内存限制）的快捷且方便的途径。

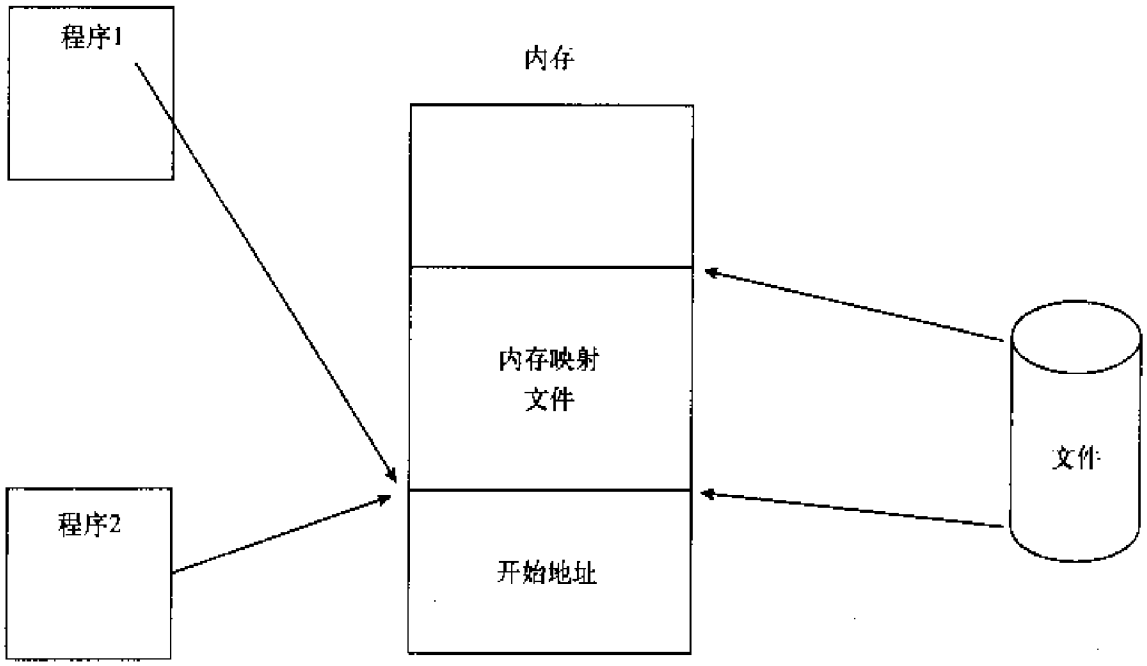



图 16-3

### 16.6.2 mmap系统调用

mmap系统调用用于创建内存映射文件。mmap系统调用的格式如下：

```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

下面是输入值：

- start：在内存中的什么位置映射文件
- length：加载到内存中的字节数量
- prot：内存保护设置
- flags：要创建的映射对象的类型
- fd：要映射到内存的文件的文件句柄
- offset：文件中要复制到内存的数据的起点

start值可以设置为0，这使系统可以选择在内存中的什么位置存放内存映射文件。如果offset值被设置为0，length值被设置为文件长度，就把整个文件映射到内存。mmap函数在很大程度上依赖于系统的内存页面长度。如果length值不能填充整个页面（或者填充多个页面，还剩下没有填充满的页面），就使用0填充页面的剩余部分。如果使用offset值，它必须是系统页面长度的倍数。

prot值包含确定对内存映射文件允许的访问权限的设置（和系统调用open中使用的访问权限

类似)。可以使用的值如下表所示。

| 保护名称 | 值 | 描 述 |
|---|---|---|
| PROT_NONE | 0 | 不允许数据访问 |
| PROT_READ | 1 | 允许读取访问 |
| PROT_WRITE | 2 | 允许写入访问 |
| PROT_EXEC | 4 | 允许执行访问 |

flag's值定义操作系统如何控制内存映射文件。可以使用很多不同的标志，但是最常用的值有两个，如下表所示。

| 标志名称 | 值 | 描 述 |
|---|---|---|
| MAP_SHARE | 1 | 和其他进程共享内存映射文件的改动 |
| MAP_PRIVATE | 2 | 保持所有改动对这个进程是私有的 |

这两种模式之间的区别很大。标志MAP_SHARE指示操作系统把对内存映射文件做出的任何改动都写入原始文件。关闭内存映射文件时，标志MAP_PRIVATE忽略对内存映射文件作出的任何改动。虽然这种方式看上去很奇怪，但是如果需要创建需要快速访问的临时文件，那么它实际上是很方便的。

对内存映射文件做出改动时，并没有同时把改动写入原始文件。这是要记住的很重要的一点。使用两个系统调用确保把内存映射文件中的数据写入原始文件:

- msync: 对原始文件和内存映射文件进行同步
- munmap: 从内存中删除内存映射文件并且把所有改动写入原始文件

如果打算在做出任何改动之后，在很长一段时间内把内存映射文件保存在内存中，那么使用系统调用msync确保把改动写入文件是个好主意。如果在发出系统调用msync或者munmap之前，程序或者操作系统崩溃，对内存映射文件做出的任何改动都不会写入原始文件。

系统调用msync和munmap具有类似的格式:

```
int msync(const void *start, size_t length, int flags);
int munmap(void *start, size_t length);
```

输入值start是内存映射文件在内存中的开始位置。系统调用mmap返回这个值。输入值length是要写入原始文件的字节数量。msync的输入值flags可以定义如何更新原始文件:

- MS_ASYNC: 在下次可以写入文件时安排更新，并且系统调用返回。
- MS_SYNC: 系统调用等待，直到做出更新，然后再返回调用程序。

要记住的重要的一点是内存映射文件不能改动原始文件的长度。

### 16.6.3 mmap汇编语言格式

在汇编语言中使用系统调用mmap需要使用第12章“使用Linux系统调用”中介绍的格式。在执行Linux系统调用之前，必须把文件unistd.h中找到的系统调用值存放到EAX寄存器中。系统调用mmap的系统调用值显示在下表中。

| 系统调用 | 值 |
|---|---|
| mmap | 90 |
| munmap | 91 |
| msync | 144 |

但是，使用第12章介绍的标准Linux系统调用格式和系统调用mmap时有一个问题。虽然大多数系统调用使用通用寄存器存放输入值，但是输入值超过5个的系统调用不能使用这种方法（没有足够的寄存器可用）。

替换的做法是，具有超过5个输入值的系统调用（比如mmap）从内存中定义的结构读取输入值。从指定的内存位置开始，依次把每个输入值存放到内存中。发出系统调用之前，将内存结构的开始位置存放在EBX寄存器中，如图16-4所示。

在函数调用之前，输入值要么存放在单独定义的内存中，要么压入一般的程序堆栈。如果把这些值压入堆栈，要记住在堆栈中按照相反的顺序存放它们（从右到左）。还必须记住当系统调用返回时从堆栈中删除它们（把ESP指针移动回去）。

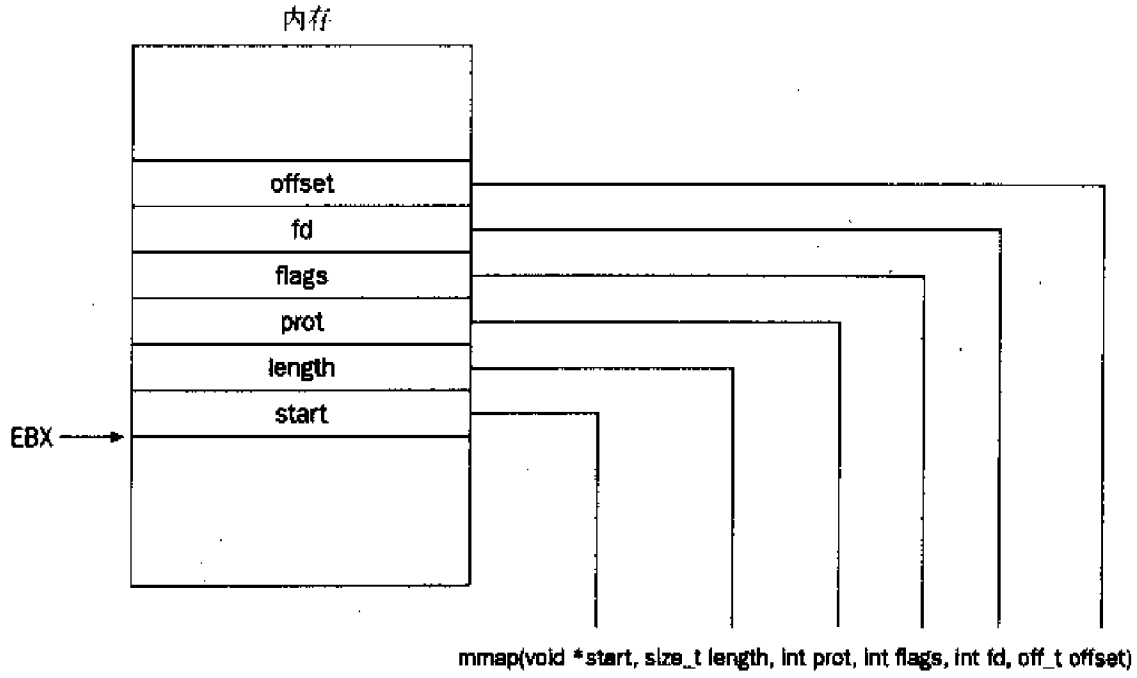



图 16-4

系统调用mmap使用的模板如下：

```
pushl $0     # offset of 0
pushl filehandle    # the file handle of the open file
pushl $1      # MAP_SHARED flag set to write changed data back to file
pushl $3      # PROT_READ and PROT_WRITE permissions
pushl size    # the size of the entire file
pushl $0      # Allow the system to select the location in memory to start
movl %esp, %ebx    # copy the parameters location to EBX
movl $90, %eax     # set the system call value
int $0x80
addl $24, %esp
movl %eax, mappedfile    # store the memory location of the memory mapped file
```

这个例子创建包含整个文件的内存映射文件，允许对文件进行读取和写入处理，并且允许把所有更新存储到原始文件中。系统调用的返回值存放在EAX寄存器中，它指向内存映射文件开始的内存位置。可以存储这个值，然后可以像使用任何其他内存位置（比如程序readtest中使用的缓冲区位置）一样使用它。

为了对内存中的文件进行unmap操作，系统调用unmap的格式如下：

```
movl $91, %eax
movl mappedfile, %ebx
movl size, %ecx
int $0x80
```

因为系统调用munmap只使用两个输入值，所以在发出系统调用之前，可以把它们存放在EBX和ECX寄存器中。如果成功地把数据写回文件，系统调用munmap将返回0值。如果不成功，就会返回-1值。

### 16.6.4 mmap范例

本节演示如何使用系统调用mmap把整个文件映射到内存中、修改内存映射文件中的数据以及把数据写回原始文件。

1. 程序的组成部分

在构建mmap主程序之前，需要构建其他几个程序片断。首先，为了使用系统调用mmap，必须知道要存放在内存中的文件的长度。可以使用几种不同方法确定文件长度。一种最简单的方法是使用Linux系统调用llseek（它是POSIX系统调用lseek的扩展）。使用Linux系统调用llseek创建一个函数，给定打开文件的文件句柄，使用这个函数计算文件长度。

其次，必须有一个函数修改内存映射文件的数据，以便我们能知道完成操作时数据被正确地存放回了文件中。程序readtest3.s中的函数convert可以很好地完成这个工作，对它做出一些修改，使它成为正确的C样式汇编语言函数。

系统调用llseek试图使文件指针前进到文件中的特定位置。可以指示系统调用llseek使指针指向文件末尾，并且查看从文件开头到这个位置有多少字节。

系统调用llseek的格式如下：

```
int _llseek(unsigned int fd, unsigned long offset_high, unsigned long offset_low,
            loff_t *result, unsigned int whence);
```

输入值fd是文件的文件句柄。输入值offset_high和offset_low定义从文件开头算起的偏移值的高位和低位部分（long类型数据）。因为我们使用系统调用确定整个文件的长度，所以把这些值设置为0。输入值result有些容易引起误解。它是将存储系统调用的结果的内存位置的地址。

输入值whence用于确定系统调用llseek将到达文件中的什么位置。我们感兴趣的值是SEEK_END，它的值为2。

函数sizefunc.c使用系统调用llseek确定文件长度：

```
# sizefunc.s - Find the size of a file
.section .text
.globl sizefunc
.type sizefunc, @function
```

```
sizefunc:
   pushl %ebp
   movl %esp, %ebp
   subl $8, %esp
   pushl %edi
   pushl %esi
   pushl %ebx
   movl $140, %eax
   movl 8(%ebp), %ebx
   movl $0, %ecx
   movl $0, %edx
   leal -8(%ebp), %esi
   movl $2, %edi
   int $0x80
   movl -8(%ebp), %eax

   popl %ebx
   popl %esi
   popl %edi
   movl %ebp, %esp
   popl %ebp
   ret
```

第一组指令执行一般的C样式函数开头功能，包括在堆栈中为局部变量保留4个字节（参见第14章）。这个局部变量用作系统调用llseek的结果值。

函数sizefunc假设被调用时，要检查的文件的文件句柄作为第一个输入值被传递进来（位置是8（%ebp））。使用LEA指令把局部变量的位置（-8（%ebp））存放到%esi寄存器中（这个局部变量必须是8字节长，因为llseek函数返回的文件长度为long类型值）。系统调用完成时，文件长度将存放在局部变量中，然后把它传送给EAX寄存器。这是返回给调用程序的值。

函数convert被修改为正确的C样式函数，程序convertit.s中显示它：

```
# convert.s - A function to convert lower case letters to upper case
.section .text
.type convert, @function
.globl convert
convert:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi

   movl 12(%ebp), %esi
   movl %esi, %edi
   movl 8(%ebp), %ecx

convert_loop:
   lodsb
   cmpb $0x61, %al
   jl skip
   cmpb $0x7a, %al
   jg skip
   subb $0x20, %al
skip:
   stosb
   loop convert_loop
```

```
pop %edi
pop %esi
movl %ebp, %esp
popl %ebp
ret
```

在标准C样式函数开头代码之后，从堆栈读取输入值。首先读取缓冲区的位置并且存放到ESI寄存器中，然后读取文件长度并且存放在ECX寄存器中。下一段执行转换功能，它遍历缓冲区，把小写ASCII字母转换为大写ASCII字母。达到缓冲区的末尾时，执行标准C样式函数结尾指令，函数返回调用程序。

2. 主程序

既然已经准备好了所需的所有代码片断，现在是编写完整的应用程序的时候了。程序fileconvert.s把文件中的文本从小写转换为大写：

```
# fileconvert.s - Memory map a file and convert it
.section .bss
   .lcomm filehandle, 4
   .lcomm size, 4
   .lcomm mappedfile, 4
.section .text
.globl _start
_start:
   # get the file name and open it in read/write mode
   movl %esp, %ebp
   movl $5, %eax
   movl 8(%ebp), %ebx
   movl $0102, %ecx
   movl $0644, %edx
   int $0x80
   test %eax, %eax
   js badfile
   movl %eax, filehandle

   # find the size of the file
   pushl filehandle
   call sizefunc
   movl %eax, size
   addl $4, %esp

   # map file to memory
   pushl $0
   pushl filehandle
   pushl $1      # MAP_SHARED
   pushl $3      # PROT_READ | PROT_WRITE
   pushl size    # file size
   pushl $0      # NULL
   movl %esp, %ebx
   movl $90, %eax
   int $0x80    test %eax, %eax
   js badfile
   movl %eax, mappedfile
   addl $24, %esp
   # convert the memory mapped file to all uppers
   pushl mappedfile
```

```
    pushl size
    call convert
    addl $8, %esp

    # use munmap to send the changes to the file
    movl $91, %eax
    movl mappedfile, %ebx
    movl size, %ecx
    int $0x80
    test %eax, %eax
    jnz badfile

  # close the open file handle
    movl $6, %eax
    movl filehandle, %ebx
    int $0x80

badfile:
    movl %eax, %ebx
    movl $1, %eax
    int $0x80
```

完整的程序使用两个辅助函数，还有系统调用mmap和munmap。应用程序代码中的每个指令段执行一个单一功能。执行步骤如下:

1) 使用读/写访问打开文件。

2) 使用函数sizefunc确定文件长度。

3) 使用系统调用mmap把文件映射到内存中。

4) 把内存映射文件的全部内容转换为大写字母。

5) 使用munmap把内存映射文件写入原始文件。

6) 关闭原始文件并且退出。

3. 监视程序

因为这个程序不需要任何C函数，所以只需汇编和连接它们:

```
$ as -o sizefunc.o sizefunc.s
$ as -o convert.o convert.s
$ as -o fileconvert.o fileconvert.s
$ ld -o fileconvert fileconvert.o sizefunc.o convert.o
$ cat cpuid.txt
The processor Vendor ID is 'GenuineIntel'
The processor Vendor ID is 'GenuineIntel'
$ ./fileconvert cpuid.txt
$ cat cpuid.txt
THE PROCESSOR VENDOR ID IS 'GENUINEINTEL'
THE PROCESSOR VENDOR ID IS 'GENUINEINTEL'
$
```

读者可以使用喜欢的任何文本文件测试程序fileconvert。要记住原始文件将被转换为大写，所以你也许希望在测试之前把文件复制到其他位置。

因为程序fileconvert使用系统调用，所以在程序运行时，可以使用程序strace（在第12章中讲过）监视发出的系统调用:

```
$ strace ./fileconvert test.txt
execve("./fileconvert", ["./fileconvert", "test.txt"], [/* 38 vars */]) = 0
open("test.txt", O_RDWR|O_CREAT, 0644)  = 3
_llseek(3, 0, [1419], SEEK_END)         = 0
old_mmap(NULL, 1419, PROT_READ|PROT_WRITE, MAP_SHARED, 3, 0) = 0x40000000
munmap(0x40000000, 1419)                = 0
close(3)                                = 0
_exit(0)                                = ?
$
```

注意，程序strace显示系统调用的C样式版本，包括程序传递给它们的输入值，还有它们生成的返回值。如果系统调用有什么问题，程序strace是进行错误查找的优秀工具。

## 16.7 小结

本章讨论在汇编语言程序中使用Linux系统调用处理文件的输入和输出。Linux提供了很多不同的系统调用，可以使用它们操作文件。在使用文件之前，必须使用系统调用open。系统调用open试图使用定义好的权限和访问类型打开系统中的一个文件。必须记住检查这个系统调用的返回值，以便确保打开文件的操作没有发生问题。

打开文件之后，可以使用系统调用write把数据写入文件。数据如何写入文件取决于系统调用open中使用的访问类型。可以把新的数据追加到已有文件的末尾，也可以使用新的数据重写已有文件。再次强调，必须注意系统调用的返回值，以便确保正确地执行了write指令。

系统调用read可以从文件读取数据并且把数据存放在内存缓冲区中。必须指定单一系统调用read从文件读取多少字节的数据。如果这个值小于文件中字节的数量，就使用文件指针跟踪系统调用read停止在了什么位置，另一次系统调用read将从文件中的这个位置开始读取数据。如果这个值大于文件中字节的数量，剩余的字节将被复制到内存缓冲区中，并且系统调用read的返回值表示实际读取了多少字节。如果返回值为0，则表示已经到达了文件末尾。

很多汇编语言程序执行复杂的操作序列，把从文件读出的数据存放到内存缓冲区中，处理数据（要么修改数据、要么扫描数据），然后把数据写到另一个文件或者STDOUT控制台输出。这样的事件序列可以整合到一个循环中，从文件读取数据块，处理它们，然后把它们写回到另一个文件。每个数据块的处理独立于其他数据块，对内存缓冲区内包含的数据块进行必要的处理。

UNIX系统提供了一个系统调用，使得可以一次把整个文件的内容读取到内存中（假设文件长度小于虚拟内存的大小），而不必手动地把数据块读取到内存缓冲区中并且处理数据。这种技术称为内存映射（memory mapping）。文件被映射到内存中之后，可以使用任意数量的不同程序扫描或者修改内存映射块内包含的数据。完成处理之后，可以选择使用修改后的内存映射文件替换原始文件。这是快速修改大型文件的优秀技术。

本书的最后一章研究IA-32的MMX和SSE程序设计这个高级领域。Intel的MMX、SSE、SSE2和SSE3技术为程序员提供快速处理大量数据的高级方法。第17章介绍这些技术，并且演示如何在汇编语言程序中使用它们。

# 第17章 使用高级IA-32特性

Intel的奔腾处理器系列给IA-32指令集提供了高级数学处理能力。如果开发需要大量数学处理（比如音频和视频处理）的应用程序，利用高级数学特性能够在很大程度上提高应用程序的性能。

本章介绍奔腾处理器上使用的单指令多数据（Single Instruction Multiple Data，SIMD）指令系列。首先，简要介绍SIMD的主要特性。之后，介绍如何使用MMX架构指令。接下来，讨论SSE特性，并且演示如何在单一指令中使用SSE处理多个数据元素。最后，介绍SSE2特性，还有在汇编语言程序中使用它们的范例。

## 17.1 SIMD简介

在第2章“IA-32平台”和第7章“使用数字”中，介绍了IA-32的SIMD架构中的各种特性和数据类型。下面进行简要回顾，IA-32的SIMD架构当前由4种技术构成：

- 多媒体扩展（Multimedia Extension，MMX）
- 流化SIMD扩展（Streaming SIMD Extension，SSE）
- 流化SIMD扩展第二实现（Streaming SIMD Extension Second Implementation，SSE2）
- 流化SIMD扩展第三实现（Streaming SIMD Extension Third Implementation，SSE3）

就像第2章中讨论的，不同的Intel奔腾处理器支持SIMD架构的不同版本。奔腾MMX和奔腾II处理器支持MMX。奔腾III处理器支持MMX和SSE技术。奔腾4处理器支持MMX、SSE和SSE2技术。奔腾4HT（超线程）和Xeon处理器支持MMX、SSE、SSE2和SSE3技术。

SIMD技术的主要好处是它为程序员提供使用单一指令执行并行数学操作的能力。MMX和SSE架构提供可以保存打包数据的附加寄存器（多个数据值在单一寄存器中）。MMX和SSE指令具有一次对寄存器中所有打包数据元素执行单一数学操作的能力。

本节简要回顾这些技术给IA-32指令集提供的特性。

### 17.1.1 MMX

MMX技术的主要目的是对integer数据类型执行SIMD操作。MMX的SIMD架构提供3种附加的整数值类型：

- 64位打包字节整数（包含8个单字节整数值）
- 64位打包字整数（包含4个字整数值）
- 64位打包双字整数（包含2个双字整数值）

因为MMX整数数据类型使用64位，所以不能使用一般的通用寄存器保存它们。替换的做法是，MMX技术利用80位FPU寄存器执行其所有的数学操作。MMX寄存器被命名为MM0到MM7。它们直接映射到FPU寄存器R0到R7。但是，和FPU寄存器不同，MMX寄存器是静态的，它们不

能作为堆栈使用。MM0寄存器用于引用FPU寄存器R0。把数据值存放在MM0寄存器中不会把前一个值移动到MM1寄存器。FPU堆栈寄存器顶部值不会对MMX指令有任何影响。

虽然FPU寄存器用于保存MMX数据，但是它们也用于保存FPU数据。这可能造成非常大的混乱。在MMX模式下使用这些寄存器处理MMX数据（寄存器的指数值设置为全1），在FPU模式下使用寄存器处理一般的FPU扩展双精度浮点数据。

不幸的是，在MMX模式下使用FPU寄存器时，FPU的标记寄存器会被破坏。为了解决这个问题，最好把使用FPU寄存器的指令和使用MMX寄存器的指令分开。解决方案是使用FSAVE或FXSAVE指令把FPU寄存器保存到内存中，当MMX指令执行完成时使用FRSTOR或FXRSTOR指令恢复寄存器。FSAVE和FRSTOR指令仅仅保存FPU状态（在第9章讨论和演示过）。FXSAVE和FXRSTOR指令保存FPU、MMX和SSE状态。

MMX操作执行完毕时，应该使用EMMS指令清除FPU的标记寄存器值，以便确保FPU指令正确地执行。

正如所见，64位打包整数值保存多个整数值。MMX架构包含附加指令，用于支持在单一指令内处理多个整数值（因此有了“单指令多数据”这个术语）。

### 17.1.2 SSE

SSE技术的主要目的是对浮点数据执行SIMD操作。SSE架构提供另一种新的数据类型：128位打包的单精度浮点数据类型。这个数据类型用于包含4个单精度浮点值（第7章中介绍过，单精度浮点值需要32位）。

因为新的数据类型需要128位，所以为进行SSE处理创建了新的寄存器集合。这些新的寄存器包括8个128位寄存器（XMM0到XMM7），使用它们保存128位打包的单精度浮点数据值。SSE浮点数学操作使用XMM寄存器。

SSE架构也包含新的指令，用于对打包的单精度浮点值执行SIMD数学操作。这样最多可以在单一操作中进行4个浮点运算。

### 17.1.3 SSE2

SSE2架构通过添加5种新的数据类型扩展了SSE核心架构：

- 128位打包双精度浮点值（包含2个双精度值）
- 128位打包字节整数值（包含16个单字节整数值）
- 128位打包字整数值（包含8个字整数值）
- 128位打包双字整数值（包含4个双字整数值）
- 128位打包四字整数值（包含2个四字整数值）

所有这些新的数据类型都使用128位XMM寄存器保存要处理的数据。SSE2技术提供附加的浮点和整数SIMD操作。它包含对打包数据执行数学运算的附加指令（同样使用单一操作）。

SSE3架构没有在SSE2架构的基础上添加任何附加的数据类型。它为SSE2数据类型的高级处理提供了新的指令。

## 17.2 检测支持的SIMD操作

在研究MMX和SSE指令之前，最好了解将运行应用程序的处理器是否支持它们。记住，不仅必须考虑处理器是否是奔腾系列，而且必须考虑它是否是Intel处理器(比如有可能是AMD处理器)。正如第2章中讨论的，大多数AMD处理器系列支持MMX指令集合，但是不支持SSE技术。本节介绍可以在汇编语言程序中使用的检测处理器是否支持MMX、SSE、SSE2或者SSE3指令的逻辑。

### 17.2.1 检测支持

CPUID指令提供识别处理器是否支持各种SIMD技术的方法。如果读者从头开始阅读本书，应该在第4章“汇编语言程序范例”中看到过CPUID指令的使用。

在那个例子中，使用CPUID指令生成处理器的厂商ID。也可以使用CPUID指令生成处理器的其他信息。这个指令生成的信息的类型由执行CPUID指令时EAX寄存器的值控制。

当EAX寄存器的值为1时，CPUID指令返回处理器签名信息。处理器签名信息包括两个包含处理器特性标志的寄存器。ECX和EDX寄存器包含的位表示不同特性在处理器上是否可用。用于表示SIMD特性的位如下表所示。

| 寄存器 | 位 | 特　性 |
|---|---|---|
| EDX | 23 | 支持MMX指令 |
| EDX | 25 | 支持SSE指令 |
| EDX | 26 | 支持SSE2指令 |
| ECX | 0 | 支持SSE3指令 |

知道在哪些寄存器中查看哪些位之后，就可以使用TEST指令把CPUID指令返回的ECX和EDX寄存器中的值和表示特性标志的设置值进行比较。如果TEST指令设置零标志为1，就表明不支持该特性。如果没有设置零标志，特性位肯定被启用了，特性肯定被支持。

为了确定放在TEST指令中的值，必须了解所需特性的位位置的十六进制值。图17-1演示如何查找这些值。

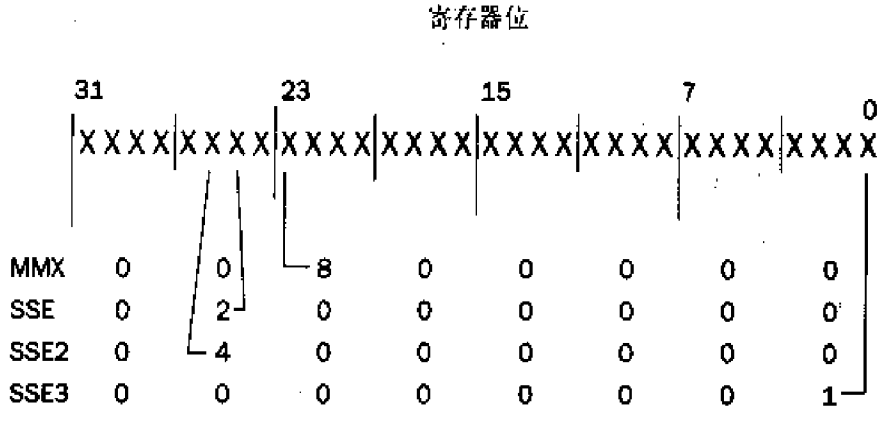



图 17-1

如图17-1所示，这样使用TEST指令检测MMX特性:

```
test $0x00800000, %edx
jz notfound
```

如果零标志被设置为1，就表示在处理器上MMX特性不可用。类似地，下面使用TEST指令检测SSE特性：

```
test $0x02000000, %edx
jz notfound
```

对于SSE2特性：

```
test $0x04000000, %edx
jz notfound
```

最后，对于SSE3特性：

```
test $0x00000001, %ecx
jz notfound
```

现在可以在汇编语言程序中使用这些测试，检测处理器上的SIMD特性。

### 17.2.2 SIMD特性程序

程序features.s使用SIMD测试列出在处理器上找到的SIMD特性：

```
# features.s - Determine MMX, SSE, SSE2, and SSE3 capabilities
.section .data
gotmmx:
   .asciz "Supports MMX"
gotsse:
   .asciz "Supports SSE"
gotsse2:
   .asciz "Supports SSE2"
gotsse3:
   .asciz "Supports SSE3"
output:
   .asciz "%s\n"
.section .bss
   .lcomm ecxdata, 4
   .lcomm edxdata, 4
.section .text
.globl _start
_start:
   nop
   movl $1, %eax
   cpuid
   movl %ecx, ecxdata
   movl %edx, edxdata

   test $0x00800000, %edx
   jz done
   pushl $gotmmx
   pushl $output
   call printf
   addl $8, %esp

   movl edxdata, %edx
```

```
    test $0x02000000, %edx
    jz done
    pushl $gotsse
    pushl $output
    call printf
    addl $8, %esp

    movl edxdata, %edx
    test $0x04000000, %edx
    jz done
    pushl $gotsse2
    pushl $output
    call printf
    addl $8, %esp

    movl ecxdata, %ecx
    test $0x00000001, %ecx
    jz done
    pushl $gotsse3
    pushl $output
    call printf
    addl $8, %esp

done:
    pushl $0
    call exit
```

程序features.s为它在处理器上检测到的每种SIMD特性显示一个文本行。首先把EAX寄存器设置为1，以便启用CPUID处理器签名模式。执行CPUID指令之后，处理器特性标志被加载到ECX和EDX寄存器。为了不丢失信息，这些值被存储在.bss段中定义的内存位置中。

因为这些特性在依次开发出来的处理器上是累积的，所以可以可靠地假设只要没有发现一个特性，那么也不会发现比它更高级的特性。利用这一概念，只要没有发现一个特性，程序features.s就退出。如果发现特性，就使用C函数printf显示简短的消息。

因为程序features.s使用C函数printf和exit，所以在运行之前，必须把它和系统中的C库文件以及动态连接器连接在一起：

```
$ as -o features.o features.s
$ ld -dynamic-linker /lib/ld-linux.so.2 -lc -o feature features.o
$ ./features
Supports MMX
Supports SSE
Supports SSE2
$
```

这个特定的处理器只支持MMX、SSE和SSE2指令。到编写本书的时候，只有带有超线程功能的奔腾4处理器支持SSE3指令。

## 17.3　使用MMX指令

为了在汇编语言程序中利用MMX架构，必须执行下列步骤：

1) 从整数值创建打包整数值。

2) 把打包整数值加载到MMX寄存器中。

3) 对打包整数值执行MMX数学操作。

4) 从MMX寄存器获得结果，存放到内存位置中。

下面几节详细介绍这些步骤。

### 17.3.1 加载和获得打包的整数值

前两个步骤和最后一个步骤——按照打包的整数格式把整数值加载到MMX寄存器以及从MMX寄存器获得打包的整数结果——已经在第7章中讨论过了。使用MOVQ指令把整数值传送进和传送出MMX寄存器。

整数值必须按照打包整数格式存放到MMX寄存器中。MMX打包整数类型可以用于8个字节整数值、4个字整数值或者2个双字整数值。可以把这些值存放在内存位置中，以便加载到MMX寄存器中，就像这样：

```
.section .data
packedvalue1:
   .byte 10, 20, -30, 40, 50, 60, -70, 80
packedvalue2:
   .short 10, 20, 30, 40
packedvalue3:
   .int 10, 20
.section .text
.globl _start
_start:
   movq packedvalue1, %mm0
   movq packedvalue2, %mm1
   movq packedvalue3, %mm2
```

当然，也可以在.bss段中为每个值保留8字节的内存，然后通过程序把值存放到内存位置中。类似地，可以使用MOVQ指令把数据从MMX寄存器传送到64位内存位置中（就像第5章“传送数据”中介绍的）。

### 17.3.2 执行MMX操作

把数据加载到MMX寄存器中之后，就可以使用单一指令对打包数据执行并行操作。利用同样存放的打包整数值，对寄存器中的每个打包整数值执行操作，如图17-2所示。

下一节介绍可以对MMX打包整数数据值执行的一些操作。

1. MMX加法和减法指令

使用通用寄存器的一般整数加法和减法是非常简单明了的操作。但是，当使用打包整数数学操作时有一个问题。

对于使用通用寄存器的一般加法和减法，如果操作出现溢出的情况，就设置EFLAGS寄存器表示出现溢出。对于打包整数值操作，同时计算多个结果值。这意味着和一般整数运算不同，单一一组标志不能表示操作的结果。

替换的做法是，使用MMX加法或者减法时，必须提前做出决定在处理器遇到操作中发生溢出的情况时应该进行什么操作。可以从执行数学操作的3种溢出方法中进行选择：

- 环绕运算
- 带符号饱和运算
- 无符号饱和运算

环绕运算的方法执行数学操作，并且允许溢出环绕数据长度。这种方法假设将在执行操作之前进行检查，确保不会出现溢出情况。如果出现了，就会截断结果值，删除所有进位值。

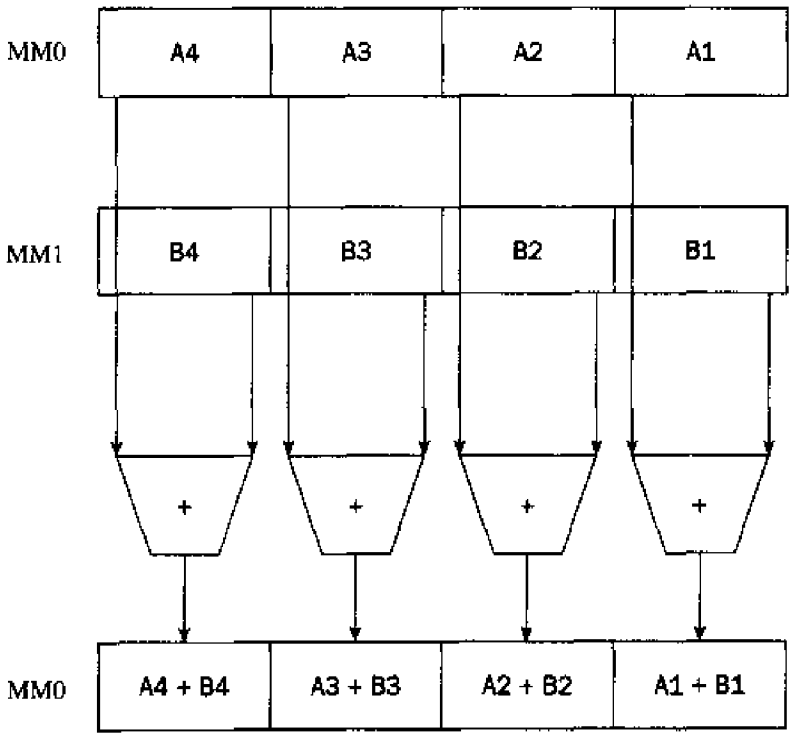



图 17-2

带符号和无符号饱和运算的方法把溢出情况的结果设置为预先设置的值，这取决于使用的打包整数值的长度，以及溢出的符号。下表介绍这些溢出值。

| 数据类型 | 正溢出值 | 负溢出值 |
|---|---|---|
| 带符号字节 | 127 | −128 |
| 带符号字 | 32 767 | −32 768 |
| 无符号字节 | 255 | 0 |
| 无符号字 | 65 535 | 0 |

如果存在正溢出情况，就把结果设置为数据类型的最大值。如果存在负溢出情况，就把结果设置为数据类型的最小值。虽然从数学运算的角度来看，饱和运算的溢出值也许没有意义，但是这样设置是有原因的。MMX技术的主要应用之一是执行图像计算以便显示图片。当计算像素的红、蓝和绿值时，正溢出应该把像素设置为最大值，就是白色。对于负溢出的情况，像素被设置为最小值，就是黑色。

决定计算需要哪种运算之后，可以根据需要执行的数学操作选择使用哪条指令。下表列出可以使用的MMX数学操作。

| MMX指令 | 描 述 |
|---|---|
| PADDB | 使用环绕的打包字节整数加法 |
| PADDW | 使用环绕的打包字整数加法 |
| PADDD | 使用环绕的打包双字整数加法 |
| PADDSB | 使用带符号饱和的打包字节整数加法 |
| PADDSW | 使用带符号饱和的打包字整数加法 |
| PADDUSB | 使用无符号饱和的字节整数加法 |
| PADDUSW | 使用无符号饱和的字整数加法 |
| PSUBB | 使用环绕的打包字节整数减法 |
| PSUBW | 使用环绕的打包字整数减法 |
| PSUBD | 使用环绕的打包双字整数减法 |
| PSUBSB | 使用带符号饱和的打包字节整数减法 |
| PSUBSW | 使用带符号饱和的打包字整数减法 |
| PSUBUSB | 使用无符号饱和的打包字节整数减法 |
| PSUBUSW | 使用无符号饱和的打包字整数减法 |

每个MMX数学操作都具有相同的格式:

```
PADDSB source, destination
```

其中source可以是MMX寄存器或者64位内存位置，destination是MMX寄存器。例如，指令

```
PADDSB %mm1, %mm0
```

把寄存器MM0的内容和MM1的内容相加，把结果存放到MM0寄存器中。

程序mmxadd.s演示简单的MMX加法:

```
# mmxadd.s - An example of performing MMX addition
.section .data
value1:
   .int 10, 20
value2:
   .int 30, 40
.section .bss
   .lcomm result, 8
.section .text
.globl _start
_start:
   nop
   movq value1, %mm0
   movq value2, %mm1
   paddd %mm1, %mm0
   movq %mm0, result

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

程序mmxadd.s通过把两个long类型的整数值存储到单一内存位置中，创建两个打包双字整数值。标签value1和value2都指向打包双字整数值。使用MOVQ指令把值加载到MMX寄存器中，然后使用PADDD指令使两个双字相加。结果存放在MM0寄存器中，然后复制到内存位置result中。

为了查看程序执行情况，必须在调试器中运行程序。把两个值加载到MMX寄存器中之后，

可以使用它们的FPU寄存器名称或者MMX寄存器名称查看它们：

```
(gdb) info all
.
.
st0            nan      (raw 0xffff000000140000000a)
st1            nan      (raw 0xffff000000280000001e)
```

ST0寄存器显示打包双字整数值。低位双字包含内存位置value1中的第一个值，高位双字包含第二个值。单步执行程序之后，可以检查内存位置：

```
(gdb) x/2d &value1
0x804909c <value1>:     10      20
(gdb) x/2d &value2
0x80490a4 <value2>:     30      40
(gdb) x/2d &result
0x80490b0 <result>:     40      60
(gdb)
```

确实，PADDD指令按照我们的期望使打包双字值相加。

2. MMX乘法指令

MMX整数乘法有些困难。因为整数乘法可能生成比输入操作数大得多的值，所以MMX乘法允许使用两条指令完成乘法操作。第一条指令PMULL把每对打包字整数值相乘，把结果的低16位存放到目标寄存器中。如图17-3所示。

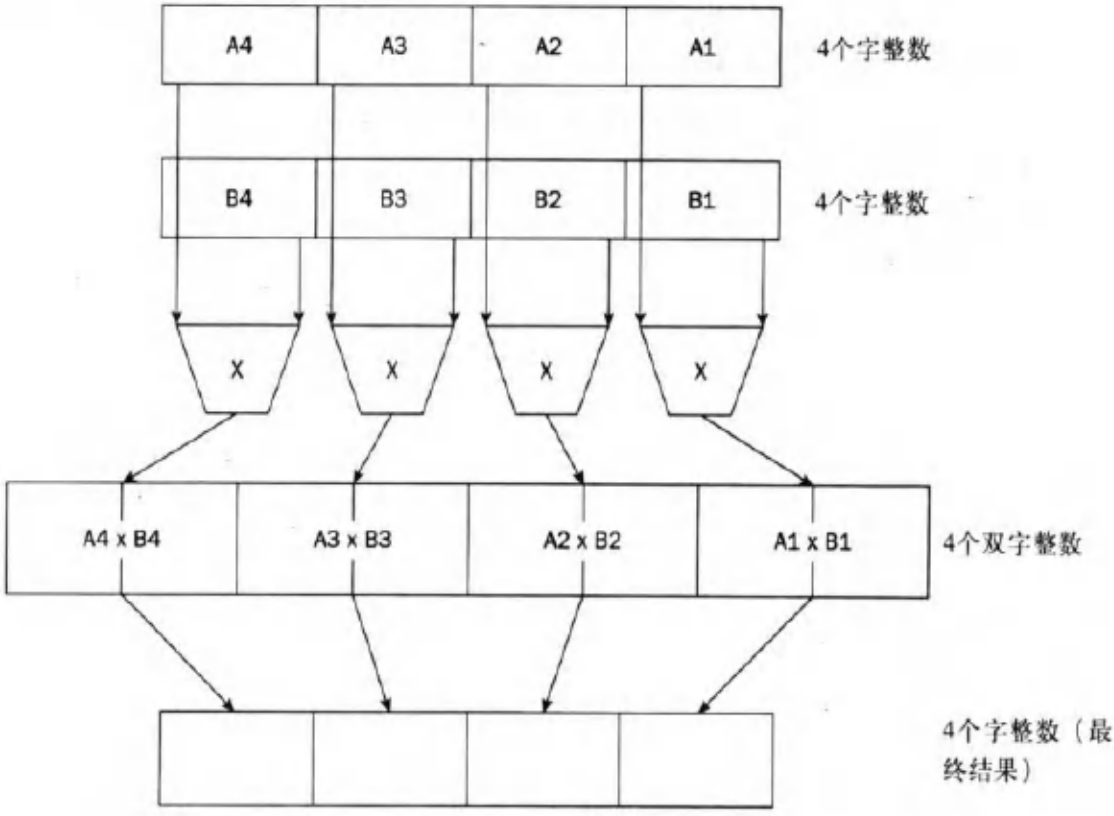



图 17-3

然后可以把结果的低16位传送到适当的内存位置，然后把原始的操作数重新加载到两个输入寄存器中。下面，使用PMULH指令，它使打包字整数值相乘，把结果的高16位存放到目标寄

存器中。现在拥有了乘法完整结果的低位和高位。

PMULL和PMULH指令有对应于带符号整数值（PMULLW和PMULHW）和无符号整数值（PMULLUW和PMULHUW）的版本。

MMX乘法系列中的一个附加指令是PMADDWD指令。PMADDWD指令是一个专用指令。它对源操作数中的4个带符号字整数值和目标操作数中的4个带符号字整数值执行乘法操作。这样生成4个带符号双字整数值。然后把相邻的双字整数值相加，生成2个双字整数结果值，如图17-4所示。

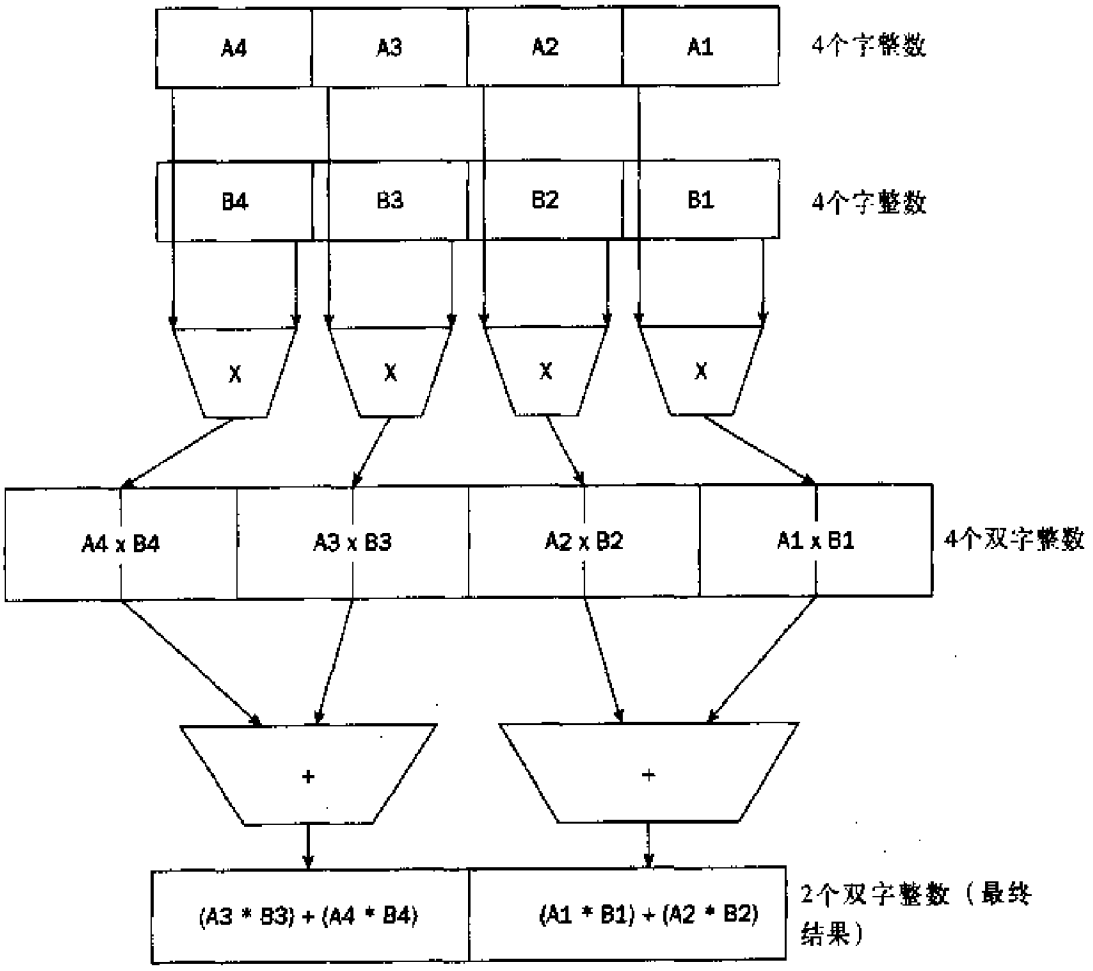



图 17-4

3. MMX逻辑和移位指令

MMX架构也提供对于四字值执行一般布尔逻辑操作和位移位操作的指令。下表列出MMX中可用的布尔逻辑指令。

| 指 令 | 描 述 |
|---|---|
| PAND | 对源和目标操作数执行按位逻辑与（AND）操作 |
| PANDN | 对目标操作数执行按位逻辑非（NOT）操作，然后对源和目标操作数执行逻辑与操作 |
| POR | 对源和目标操作数执行按位逻辑或（OR）操作 |
| PXOR | 对源和目标操作数执行按位逻辑异或（XOR）操作 |
| PSLL | 对操作数执行逻辑左移位操作，使用0填充空位 |
| PSRA | 对操作数执行逻辑右移位操作，使用0填充空位 |

逻辑指令的格式如下:

```
PAND source, destination
```

其中source可以是MMX寄存器或者64位的内存位置，destination必须是MMX寄存器。左移位指令可以使用字、双字或者四字操作数；还有要移位的位置数量；右移位指令可以使用字或者双字操作数，还有要移位的位置数量。

4. MMX比较指令

MMX架构也包含用于比较两个值的比较指令。下表介绍这些指令。

| 指　令 | 描　述 |
|---|---|
| PCMPEQB | 比较打包字节整数值的相等性 |
| PCMPEQW | 比较打包字整数值的相等性 |
| PCMPEQD | 比较打包双字整数值的相等性 |
| PCMPGTB | 判断打包字节整数值是否大于另一个 |
| PCMPGTW | 判断打包字整数值是否大于另一个 |
| PCMPGTD | 判断打包双字整数值是否大于另一个 |

MMX比较指令和一般的CMP指令有些不同。因为MMX数据类型保存多个值，所以不为相等或者大于或者小于设置标志。

替换的做法是，源和目标打包整数值进行比较，把结果存放到目标打包整数值中。如果打包整数值对满足比较条件（要么相等，要么目标值大于源值），就把结果打包整数值设置为全1。如果不满足条件，就把结果值设置为全0，如图17-5所示。

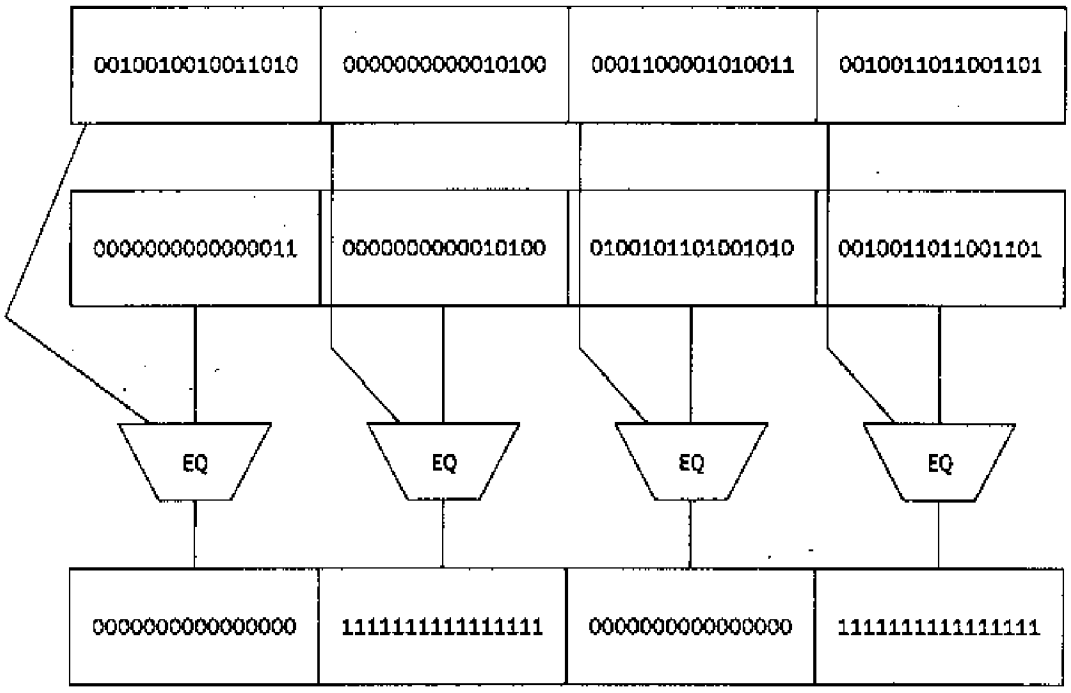



图 17-5

因为第二对和第三对打包字整数值相等，所以结果中的这些值被设置为1。其他两个打包字整数值被设置为0。

程序mmxcomp.s演示如何进行:

```
# mmxcomp.s - An example of performing MMX comparison
.section .data
value1:
   .short 10, 20, -30, 40
value2:
   .short 10, 40, -30, 45
.section .bss
   .lcomm result, 8
   .section .text
   .globl _start
   _start:
      nop
      movq value1, %mm0
      movq value2, %mm1
      pcmpeqw %mm1, %mm0
      movq %mm0, result

      movl $1, %eax
      movl $0, %ebx
      int $0x80
```

内存位置value1和value2被设置为保存4个short类型的整数值（字整数）。把它们加载到MMX寄存器中，然后使用PCMPEQW指令比较打包整数值中的4个字值。结果存放在MM0寄存器中，然后传送到内存位置result中。

必须使用调试器查看运行程序时的执行情况:

```
$ gdb -q mmxcomp
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file mmxcomp.s, line 13.
(gdb) run
Starting program: /home/rich/palp/chap17/mmxcomp

Breakpoint 1, _start () at mmxcomp.s:13
13          movq value1, %mm0
Current language:  auto; currently asm
(gdb) s
14          movq value2, %mm1
(gdb) s
15          pcmpeqw %mm1, %mm0
(gdb) s
16          movq %mm0, result
(gdb) s
18          movl $1, %eax
(gdb) x/x &value1
0x804909c <value1>:     0x0014000a
(gdb) x/x &value2
0x80490a4 <value2>:     0x0028000a
(gdb) x/x &result
0x80490b0 <result>:     0x0000ffff
(gdb)
```

两个打包双字整数值被加载到两个单独的MMX寄存器中，然后执行PCMPEQW指令。结果存放在内存位置result中。通过查看这个内存位置中的十六进制值，可以发现，对于相等的打包

整数值，结果包含FFFF，而对于不相等的打包整数值，结果包含值0000。

## 17.4 使用SSE指令

SSE架构提供对打包单精度浮点值的SIMD支持。和MMX架构一样，SSE提供新的指令，用于把数据传送到XMM寄存器中、对SSE数据执行数学操作以及从XMM寄存器获得数据。

SSE和MMX的一个区别是每个SSE指令都有两个版本。第一个版本使用后缀PS。这些指令对打包单精度浮点值执行类似于MMX操作的运算操作——打包数据值中的每个值都参与操作，并且产生的打包值包含每个打包值操作的结果（如前面的图17-2所示）。

运算指令的第二个版本使用后缀SS。这些指令对一个标量单精度浮点值执行运算操作。这些指令不对打包值中的所有浮点值执行操作，而只对打包值中的低位双字执行。源操作数中剩余的3个值直接传送给结果，如图17-6所示。

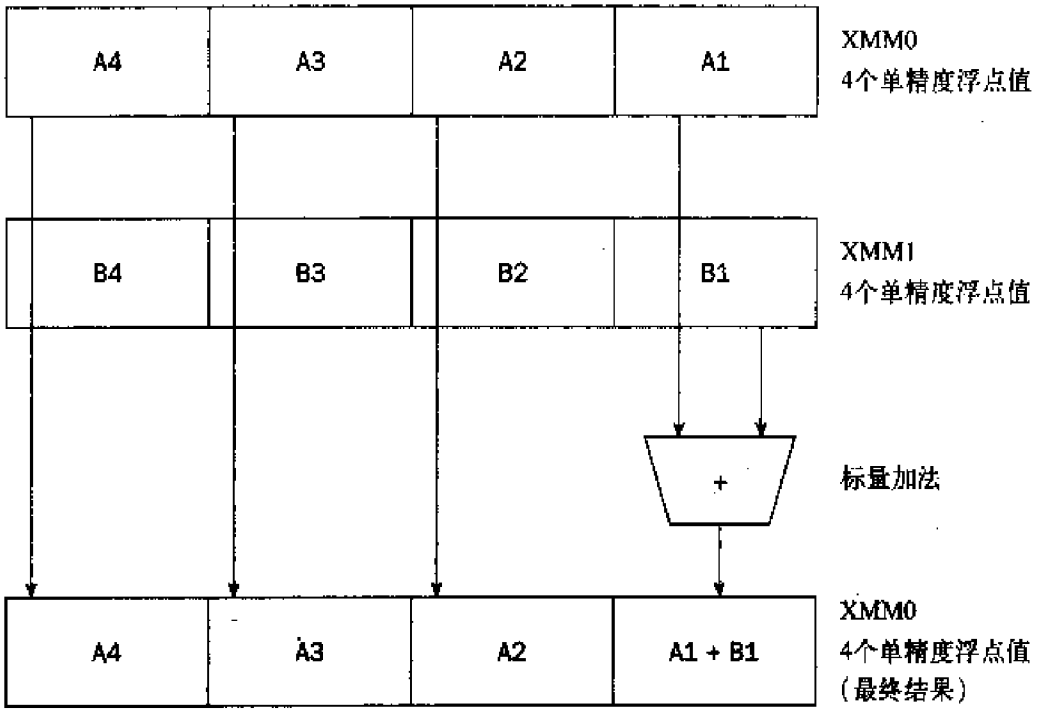



图 17-6

标量操作允许对XMM寄存器中的一个单精度浮点值执行一般的FPU类型的运算操作。

本节介绍如何在汇编语言程序中使用SSE指令来帮助提高性能。

### 17.4.1 传送数据

第7章讲解了如何把打包单精度浮点值数据传送到XMM寄存器中。为了进行回顾，下表列出了这些指令。

从下表可以看出，传送单精度浮点值在很大程度上依赖于值是否在内存中对准了。MOVAPS指令要求数据在内存中对准16字节边界。这使处理器更容易在单一操作中读取数据。

第7章中的程序ssefloat.s使用MOVUPS指令把不对准的数据传送到XMM寄存器中。如果试图在这里使用MOVAPS指令，就会在运行程序时出现分段错误。但是，可以强制汇编器对准存

储在内存中的数据。

| 指 令 | 描 述 |
|---|---|
| MOVAPS | 把4个对准的单精度值传送到XMM寄存器或者内存 |
| MOVUPS | 把4个不对准的单精度值传送到XMM寄存器或者内存 |
| MOVSS | 把1个单精度值传送到内存或者寄存器的低位双字 |
| MOVLPS | 把2个单精度值传送到内存或者寄存器的低四字 |
| MOVHPS | 把2个单精度值传送到内存或者寄存器的高四字 |
| MOVLHPS | 把2个单精度值从低四字传送到高四字 |
| MOVHLPS | 把2个单精度值从高四字传送到低四字 |

.align命令指示gas汇编器把数据对准特定的内存边界。它使用单一操作数——数据要对准的内存边界的长度。SSE的MOVAPS指令期望数据位于16字节的内存边界，所以指令应该如下：

```
.section .data
.align 16
value1:
   .float 12.34, 2345.543, -3493.2, 0.4491
.section .text
.globl _start
_start:
   movaps value1, %xmm0
```

既然数据对准了16字节边界，MOVAPS指令将正确地把内存值加载到寄存器XMM0中，且不会导致分段错误。

### 17.4.2 处理数据

SSE架构提供很多附加指令，用于处理打包单精度浮点值。本节介绍一些指令以及如何使用它们。

1. 运算指令

可以对XMM寄存器中的数据进行很多运算操作。下表列出可用的运算指令（使用打包单精度浮点值术语）。

| 指 令 | 描 述 |
|---|---|
| ADDPS | 将两个打包值相加 |
| SUBPS | 将两个打包值相减 |
| MULPS | 将两个打包值相乘 |
| DIVPS | 将两个打包值相除 |
| RCPPS | 计算打包值的倒数 |
| SQRTPS | 计算打包值的平方根 |
| RSQRTPS | 计算打包值的平方根倒数 |
| MAXPS | 计算两个打包值中的最大值 |
| MINPS | 计算两个打包值中的最小值 |
| ANDPS | 计算两个打包值的按位逻辑与（AND） |
| ANDNPS | 计算两个打包值的按位逻辑与非（AND NOT） |
| ORPS | 计算两个打包值的按位逻辑或（OR） |
| XORPS | 计算两个打包值的按位逻辑异或（XOR） |

这些指令都使用两个操作数：源操作数，它可以是128位内存或者XMM寄存器；目标操作数，它必须是XMM寄存器。

程序ssemath.s演示其中一些指令的使用：

```
# ssemath.s - An example of using SSE arithmetic instructions
.section .data
.align 16
value1:
   .float 12.34, 2345., -93.2, 10.44
value2:
   .float 39.234, 21.4, 100.94, 10.56
.section .bss
   .lcomm result, 16
.section .text
.globl _start
_start:
   nop
   movaps value1, %xmm0
   movaps value2, %xmm1

   addps %xmm1, %xmm0
   sqrtps %xmm0, %xmm0
   maxps %xmm1, %xmm0
   movaps %xmm0, result

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

程序ssemath.s把打包单精度浮点值集加载到XMM寄存器中，并且执行一些基本的运算操作。寄存器XMM0中的结果被传送回内存中使用标签result标记的位置中。汇编和连接程序之后，可以在调试器中运行它并且监视结果：

```
$ gdb -q ssemath
(gdb) break *_start+1
Breakpoint 1 at 0x8048075: file ssemath.s, line 14.
(gdb) run
Starting program: /home/rich/palp/chap17/ssemath

Breakpoint 1, _start () at ssemath.s:14
14          movaps value1, %xmm0
Current language:  auto; currently asm
(gdb) s
15          movaps value2, %xmm1
(gdb) s
17          addps %xmm1, %xmm0
(gdb) print $xmm0
$1 = {f = {12.3400002, 2345, -93.1999969, 10.4399996}}
(gdb) print $xmm1
$2 = {f = {39.2340012, 21.3999996, 100.940002, 10.5600004}}
```

MOVAPS指令执行之后，两个XMM寄存器包含打包单精度浮点值。接下来，运行ADDPS指令，并且显示寄存器XMM0的值：

```
(gdb) s
```

```
18          sqrtps %xmm0, %xmm0
(gdb) print $xmm0
$3 = {f = {51.5740013, 2366.3999, 7.74000549, 21}}
```

两个打包值相加，结果存放在寄存器XMM0中。接下来，执行SQRTPS指令：

```
(gdb) s
19          maxps %xmm1, %xmm0
(gdb) print $xmm0
$4 = {f = {7.18150425, 48.6456566, 2.78208661, 4.5825758}}
```

使用单一指令计算所有4个值的平方根。接下来，确定两个打包值中的最大值，并且把它传送到内存位置result：

```
(gdb) s
20          movaps %xmm0, result
(gdb) print $xmm0
$5 = {f = {39.2340012, 48.6456566, 100.940002, 10.5600004}}
(gdb) s
22          movl $1, %eax
(gdb) x/4f &result
0x80490c0 <result>:     39.2340012      48.6456566      100.940002      10.5600004
(gdb)
```

内存位置result中的单精度浮点值与两个打包单精度浮点值中的最大值一致。注意每个最大值都是根据与其对应的打包值确定的。

2. 比较指令

和MMX的比较指令类似，SSE的比较指令单独地比较128位打包单精度浮点值的每个元素。结果是一个掩码，满足比较条件的结果是全1的值，不满足条件的结果是全0的值（记住这些指令的标量版本只对最低的双字值执行）。

下表列出可用的比较指令。

| 指 令 | 描 述 |
|---|---|
| CMPPS | 比较打包值 |
| CMPSS | 比较标量值 |
| COMISS | 比较标量值并且设置EFLAGS寄存器 |
| UCOMISS | 比较标量值（包括非法值）并且设置EFLAGS寄存器 |

UCOMISS指令可以比较包含特殊值（比如非数字（not-a-number，NaN）值）的单精度浮点值。

CMPPS指令有些不同。它可以使用所有比较类型（等于、大于、小于等等）比较两个打包值。基本的CMPPS指令有3个操作数：

```
CMPPS imp, source, destination
```

需要source和destination操作数（目标操作数必须是XMM寄存器，但是源操作数可以是128位内存位置或者XMM寄存器）。奇怪的地方是imp操作数。这个操作数称为实现操作数（implementation operand），它确定指令将执行什么类型的比较。它是一个无符号整数值，可以

把它编码为立即值或者变量。这个操作数的值确定比较类型，如下表所示。

| 执行值 | 描　述 |
|---|---|
| 0 | 等于 |
| 1 | 小于 |
| 2 | 小于或等于 |
| 3 | 无序 |
| 4 | 不等于 |
| 5 | 不小于 |
| 6 | 不小于或等于 |
| 7 | 有序 |

因此，为了确定两个XMM寄存器是否相等，可以使用下面的指令：

```
CMPPS $0, %xmm1, %xmm0
```

结果将是位掩码（全1表示值相等，全0表示值不相等），存放在寄存器XMM0中。

读者可能不熟悉有序和无序比较。它们用于发现不是合法的浮点数字表示的数据值（在第7章“使用数字”中讲解过）。当至少一个值不是合法浮点数字时，无序比较结果为真。仅当两个操作数都是合法浮点数字时，有序比较结果为真。

汇编器gas提供了替代implementation操作数的伪指令。可以不使用带有implementation操作数的CMPPS指令，而是使用只带有source和destination操作数的伪指令。伪指令显示在下表中。

| 伪指令 | 描　述 |
|---|---|
| CMPEQPS | 等于 |
| CMPLTPS | 小于 |
| CMPLEPS | 小于或等于 |
| CMPUORDPS | 无序 |
| CMPNEQPS | 不等于 |
| CMPNLTPS | 不小于 |
| CMPNLEPS | 不小于或等于 |
| CMPORDPS | 有序 |

程序ssecomp.s演示如何使用CMPEQPS指令：

```
# ssecomp.s - An example of using SSE comparison instructions
.section .data
.align 16
value1:
   .float 12.34, 2345., -93.2, 10.44
value2:
   .float 12.34, 21.4, -93.2, 10.45
.section .bss
   .lcomm result, 16
.section .text
.globl _start
_start:
   nop
   movaps value1, %xmm0
```

```
movaps value2, %xmm1

cmpeqps %xmm1, %xmm0
movaps %xmm0, result

movl $1, %eax
movl $0, %ebx
int $0x80
```

程序ssecomp.s把两个打包单精度浮点值加载到XMM寄存器中，然后使用CMPEQPS指令比较它们。结果传送到内存位置result中。

汇编和连接程序之后，在调试器中运行它并且查看运行情况：

```
(gdb) x/4x &result
0x80490c0 <result>:  0xffffffff   0x00000000   0xffffffff   0x00000000
(gdb) print $xmm0
$1 = {f = {-NaN(0x7fffff), 0, -NaN(0x7fffff), 0}}
(gdb)
```

结果显示打包值中的第一个和第三个值相等。

3. SSE整数指令

SSE架构还提供处理64位打包整数值的一些扩展特性，它们扩展了MMX提供的功能。下表介绍这些指令，它们对位于MMX寄存器中的数据执行操作。

| 指 令 | 描 述 |
|---|---|
| PAVGB | 计算打包无符号字节整数的平均值 |
| PAVGW | 计算打包无符号字整数的平均值 |
| PEXTRW | 把一个字从MMX寄存器复制到通用寄存器 |
| PINSRW | 把一个字从通用寄存器复制到MMX寄存器 |
| PMAXUB | 计算打包无符号字节整数的最大值 |
| PMAXSW | 计算打包带符号字整数的最大值 |
| PMINUB | 计算打包无符号字节整数的最小值 |
| PMINSW | 计算打包带符号字整数的最小值 |
| PMULHUW | 将打包无符号字整数相乘并且存储高位结果 |
| PSADBW | 计算无符号字节整数的绝对差的总和 |

## 17.5 使用SSE2指令

通过提供对打包双精度浮点值和128位打包整数值执行数学操作的指令，SSE2架构扩展了SSE指令。

SSE2指令使用128位XMM寄存器保存2个双精度浮点值、4个双字整数值或者2个四字整数值。提供的SSE2指令可以对所有这些数据类型执行数学操作。双精度浮点值操作可以是打包的，也可以是标量的（和SSE指令类似）。

下面几节介绍如何在汇编语言程序中使用SSE2指令。

### 17.5.1 传送数据

在第7章中介绍过，SSE2架构提供5条指令，用于在内存和XMM寄存器之间传送双精度浮点

值。下表介绍这些指令。

| 指　令 | 描　述 |
|---|---|
| MOVAPD | 把2个对准的双精度值传送到XMM寄存器或者内存 |
| MOVUPD | 把2个不对准的双精度值传送到XMM寄存器或者内存 |
| MOVDQA | 把2个对准的四字整数值传送到XMM寄存器或者内存 |
| MOVDQU | 把2个不对准的四字整数值传送到XMM寄存器或者内存 |
| MOVSD | 把1个双精度值传送到内存或者寄存器的低位四字 |
| MOVHPD | 把1个双精度值传送到内存或者寄存器的高位四字 |
| MOVLPD | 把1个双精度值传送到内存或者寄存器的低位四字 |

和SSE的数据传送指令类似，SSE2指令提供传送对准和不对准的数据的功能。为了使用MOVAPD和MOVDQA指令，内存中的数据必须位于16字节的边界。

为了对准数据，必须使用.align命令:

```
.section .data
.align 16
packedvalue1:
   .double 10.235, 289.1
packedvalue2:
   .int 10, 20, 30, 40
.section .text
.globl _start
_start:
   movapd packedvalue1, %xmm0
   movdqa packedvalue2, %xmm1
```

### 17.5.2 处理数据

SSE2指令集提供用于处理打包双精度浮点值、打包字整数值、打包双字整数值和打包四字整数值的数学指令。对于每种数学操作指令，每种数据类型都具有其自己的指令代码，下表介绍SSE2加法指令。

| 指　令 | 描　述 |
|---|---|
| ADDPD | 将打包双精度浮点值相加 |
| ADDSD | 将标量双精度浮点值相加 |
| PADDSB | 将打包带符号字节整数值相加 |
| PADDSW | 将打包带符号字整数值相加 |
| PADDD | 将打包双字整数值相加 |
| PADDQ | 将打包四字整数值相加 |

可以看出，每种数学操作的指令列表很长。这些选项也存在于乘法和除法操作中（MULPD、MULSD、DIVPD、DIVSD等等）。

和SSE指令集一样，SSE2指令集也提供专门的数学操作: SQRT、MAX和MIN。

程序sse2math.s演示如何使用这些函数:

```
# sse2math.s - An example of using SSE2 arithmetic instructions
```

```
.section .data
.align 16
value1:
   .double 10.42, -5.330
value2:
   .double 4.25, 2.10
value3:
   .int 10, 20, 30, 40
value4:
   .int 5, 15, 25, 35
.section .bss
   .lcomm result1, 16
   .lcomm result2, 16
.section .text
.globl _start
_start:
   nop
   movapd value1, %xmm0
   movapd value2, %xmm1
   movdqa value3, %xmm2
   movdqa value4, %xmm3

   mulpd %xmm1, %xmm0
   paddd %xmm3, %xmm2

   movapd %xmm0, result1
   movdqa %xmm2, result2

   movl $1, %eax
   movl $0, %ebx
   int $0x80
```

因为程序sse2math.s使用MOVAPD和MOVDQA指令加载XMM寄存器，所以.data段必须对准16字节边界。这需要使用.align命令。把数据加载到XMM寄存器中之后，使用SSE2的MULPD指令执行2个双精度浮点值的乘法操作，使用PADDD指令执行4个双字整数值的加法操作。

汇编和连接程序之后，可以在调试器中监视程序的运行情况，以便确保程序正常工作。首先，在加载4个XMM寄存器之后停止程序，并且查看XMM寄存器：

```
(gdb) print $xmm0
$1 = {v4_float = {0.0587499999, 2.57562494, -7.46297859e-36, -2.33312488},
  v2_double = {10.42, -5.3300000000000001},
  v16_int8 = "xxp=\nx$@R,\036\205@Q\025x", v8_int16 = {-23593, 15728, -10486,
    16420, -18350, -31458, 20971, -16363}, v4_int32 = {1030792151, 1076156170,
    -2061584302, -1072344597}, v2_int64 = {4622055556569408471,
    -4605684971923916718}, uint128 = 0xc01551eb851eb8524024d70a3d70a3d7}
(gdb) print $xmm1
$2 = {v4_float = {0, 2.265625, -107374184, 2.01249981}, v2_double = {4.25,
    2.1000000000000001},
  v16_int8 = "\000\000\000\000\000\000\021@xxxxxx\000@", v8_int16 = {0, 0, 0,
    16401, -13107, -13108, -13108, 16384}, v4_int32 = {0, 1074855936,
    -858993459, 1073794252}, v2_int64 = {4616471093031469056,
    4611911198408756429}, uint128 = 0x4000cccccccccccd4011000000000000}
(gdb) print $xmm2
$3 = {v4_float = {1.40129846e-44, 2.80259693e-44, 4.20389539e-44,
    5.60519386e-44}, v2_double = {4.2439915824246103e-313,
```

```
    8.4879831653432862e-313},
  v16_int8 = "\n\000\000\000\024\000\000\000\036\000\000\000(\000\000",
  v8_int16 = {10, 0, 20, 0, 30, 0, 40, 0}, v4_int32 = {10, 20, 30, 40},
  v2_int64 = {85899345930, 171798691870},
  uint128 = 0x000000280000001e000000140000000a}
(gdb) print $xmm3
$4 = {v4_float = {7.00649232e-45, 2.1019477e-44, 3.50324616e-44,
    4.90454463e-44}, v2_double = {3.1829936866949413e-313,
    7.4269852696136172e-313},
  v16_int8 = "\005\000\000\000\017\000\000\000\031\000\000\000#\000\000",
  v8_int16 = {5, 0, 15, 0, 25, 0, 35, 0}, v4_int32 = {5, 15, 25, 35},
  v2_int64 = {64424509445, 150323855385},
  uint128 = 0x00000023000000190000000f00000005}
(gdb)
```

在调试器中输出XMM寄存器的值时要注意，调试器不知道加载到寄存器中的数据是什么类型，所以它按照所有可能的格式提供显示。寄存器XMM0和XMM1的数据使用v2_double格式，寄存器XMM2和XMM3的数据按照v4_int32格式。

单步执行到SSE2的数学指令和MOV指令之后，可以使用适当的数据格式显示存储在内存位置中的结果：

```
(gdb) x/2gf &result1
0x8049100 <result1>:    44.284999999999997      -11.193000000000001
(gdb) x/4wd &result2
0x8049110 <result2>:    15      35      55      75
(gdb)
```

输出显示所有值的计算结果确实正确，并且结果被存储到了内存位置中。

## 17.6 SSE3指令

SSE3架构没有提供任何新的数据类型，仅仅添加了几条指令，它们用于帮助更快地执行标准函数，或者是提供新的指令。

下面是提供的SSE3指令：

- FISTTP：把第1个FPU寄存器的值转换为整数（使用舍入）并且从FPU堆栈弹出它。
- LDDQU：快速地从内存加载128位不对准的数据值。
- MOVSHDUP：传送128位值，复制第2个和第4个32位数据元素。
- MOVSLDUP：传送128位值，复制第1个和第3个32位数据元素。
- MOVDDUP：传送64位值，复制值，使之成为128位值。
- ADDSUBPS：对于打包单精度浮点值，对第2个和第4个32位值执行加法操作，对第1个和第3个32位值执行减法操作。
- ADDSUBPD：对于打包双精度浮点值，对第2对64位值执行加法操作，对第1对执行减法操作。
- HADDPS：对操作数的相邻的数据元素执行单精度浮点加法操作。
- HADDPD：对操作数的相邻的数据元素执行双精度浮点加法操作。
- HSUBPS：对操作数的相邻的数据元素执行单精度浮点减法操作。

• HSUBPD：对操作数的相邻的数据元素执行双精度浮点减法操作。

## 17.7 小结

对于需要对数字数组执行很多数学操作的程序，试图使用标准的处理器数学函数循环遍历数组可能会消耗很多时间。新型的IA-32平台芯片为程序员提供了帮助加快速度的选择。

IA-32平台的单指令多数据（Single Instruction，Multiple Data，SIMD）技术提供指令和新的寄存器，用于使用单一指令并行地处理数据数组。多媒体扩展（Multimedia Extension，MMX）使用单一指令处理打包整数值。MMX架构使用FPU寄存器提供8个64位MMX寄存器，用于保存打包字节、字和双字整数值。数据值可以从内存位置加载到MMX寄存器中，在MMX寄存器中进行处理，然后存储回内存位置中。MMX指令集提供对打包整数值执行一般的数学运算、布尔和比较操作的指令。这使程序员可以同时处理2个双字整数、4个字整数或者8个字节整数值。

流化SIMD扩展（Streaming SIMD Extension，SSE）提供全新的一组寄存器，用于处理128位打包数据。使用8个128位XMM寄存器处理SSE数据类型。SSE数据类型提供打包单精度浮点值。这允许使用单一数学指令处理4个单精度浮点数据值。

和MMX指令类似，SSE指令提供了执行数学运算、布尔和比较操作指令的方法。SSE的比较指令有些奇怪，因为只提供了一条真正的指令。CMPPS指令使用implementation操作数确定在两个输入操作数值之间进行什么类型的比较。大多数汇编器（包括gas）使用伪指令替换CMPPS指令和implementation操作数。

除了对打包单精度浮点值进行操作之外，SSE架构还提供处理标量运算操作的指令。标量运算只对寄存器中的低位双字值执行运算功能。其他值复制source操作数中的条目。

SSE2架构提供附加指令，用于对附加的打包数据类型执行数学操作。SSE2包含处理打包双精度浮点值（每个数据元素2个值）以及打包字节、打包字、打包双字和打包四字整数值的指令。这为程序员提供了大量可以利用的处理能力。每种打包数据类型都包含数学指令，用于使用单一指令并行地处理打包数据类型。

SSE3架构没有提供任何新的数据类型，但是提供了若干附加的指令，用于处理XMM寄存器中的数据。还有若干指令用于执行混合的数学操作，比如对寄存器内的元素执行组合的加法和减法操作。这些指令用于执行视频和音频数据处理中经常使用的特殊功能。

本书的内容到此结束，但是对汇编语言的学习不应该到此为止。我们希望本书能够帮助读者开始把汇编语言程序和高级语言程序一起使用，以便优化应用程序，但是要学习的东西总是很多。随着新的处理器面世，就会添加新的优化特性。我强烈建议经常浏览各个芯片厂商的Web站点。它们提供优秀的文档，介绍新的处理器中可用的新指令和数据类型。我希望本书带给读者愉快的阅读经历，并且希望读者能够把新的知识应用到专业程序设计生涯中。

• HSUBPD：对操作数的相邻的数据元素执行双精度浮点减法操作。

## 17.7 小结

对于需要对数字数组执行很多数学操作的程序，试图使用标准的处理器数学函数循环遍历数组可能会消耗很多时间。新型的IA-32平台芯片为程序员提供了帮助加快速度的选择。

IA-32平台的单指令多数据（Single Instruction，Multiple Data，SIMD）技术提供指令和新的寄存器，用于使用单一指令并行地处理数据数组。多媒体扩展（Multimedia Extension，MMX）使用单一指令处理打包整数值。MMX架构使用FPU寄存器提供8个64位MMX寄存器，用于保存打包字节、字和双字整数值。数据值可以从内存位置加载到MMX寄存器中，在MMX寄存器中进行处理，然后存储回内存位置中。MMX指令集提供对打包整数值执行一般的数学运算、布尔和比较操作的指令。这使程序员可以同时处理2个双字整数、4个字整数或者8个字节整数值。

流化SIMD扩展（Streaming SIMD Extension，SSE）提供全新的一组寄存器，用于处理128位打包数据。使用8个128位XMM寄存器处理SSE数据类型。SSE数据类型提供打包单精度浮点值。这允许使用单一数学指令处理4个单精度浮点数据值。

和MMX指令类似，SSE指令提供了执行数学运算、布尔和比较操作指令的方法。SSE的比较指令有些奇怪，因为只提供了一条真正的指令。CMPPS指令使用implementation操作数确定在两个输入操作数值之间进行什么类型的比较。大多数汇编器（包括gas）使用伪指令替换CMPPS指令和implementation操作数。

除了对打包单精度浮点值进行操作之外，SSE架构还提供处理标量运算操作的指令。标量运算只对寄存器中的低位双字值执行运算功能。其他值复制source操作数中的条目。

SSE2架构提供附加指令，用于对附加的打包数据类型执行数学操作。SSE2包含处理打包双精度浮点值（每个数据元素2个值）以及打包字节、打包字、打包双字和打包四字整数值的指令。这为程序员提供了大量可以利用的处理能力。每种打包数据类型都包含数学指令，用于使用单一指令并行地处理打包数据类型。

SSE3架构没有提供任何新的数据类型，但是提供了若干附加的指令，用于处理XMM寄存器中的数据。还有若干指令用于执行混合的数学操作，比如对寄存器内的元素执行组合的加法和减法操作。这些指令用于执行视频和音频数据处理中经常使用的特殊功能。

本书的内容到此结束，但是对汇编语言的学习不应该到此为止。我们希望本书能够帮助读者开始把汇编语言程序和高级语言程序一起使用，以便优化应用程序，但是要学习的东西总是很多。随着新的处理器面世，就会添加新的优化特性。我强烈建议经常浏览各个芯片厂商的Web站点。它们提供优秀的文档，介绍新的处理器中可用的新指令和数据类型。我希望本书带给读者愉快的阅读经历，并且希望读者能够把新的知识应用到专业程序设计生涯中。